Model Checking Timed Automata

Techniques and Applications

Martijn Hendriks

ISBN 90-9020378-8
IPA dissertation series 2006-06

Typeset with LaTeX2ε
Printed by Print Partners Ipskamp, Enschede

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics). It has been supported by the European Community Project IST-2001-35304 AMETIST.

Model Checking Timed Automata

Techniques and Applications

een wetenschappelijke proeve op het gebied

van de Natuurwetenschappen, Wiskunde en Informatica

Proefschrift

ter verkrijging van de graad van doctor

aan de Radboud Universiteit Nijmegen

op gezag van de Rector Magnificus prof. dr. C.W.P.M. Blom,

volgens besluit van het College van Decanen

in het openbaar te verdedigen op dinsdag 4 april 2006

des namiddags om 3.30 uur precies

door

Martijn Hendriks

geboren op 20 augustus 1976

te Canberra, Australië

Promotor:

Prof. dr. Frits W. Vaandrager

Manuscriptcommissie:

Prof. dr. Rajeev Alur, University of Pennsylvania, United States
Prof. dr. ir. Joost-Pieter Katoen, RWTH Aachen University, Germany
Dr. Sergio Yovine, CNRS, France

Preface

First of all, I thank Frits Vaandrager and Jozef Hooman for convincing me to work on a PhD project and for providing a very free and stimulating atmosphere for research: I have learned a lot these last four years!

For most of my time as a PhD student I have been working in the AMETIST project which gave me the opportunity to meet and interact with a whole group of international researchers during the bi- or even tri-annual meetings in nice places. I thank all AMETIST people for these inspiring meetings. The research papers constituting this thesis originate nearly all from the AMETIST project and would not have existed without my co-authors. Gerd Behrmann, Ed Brinksma, Ling Cheung, Kim Larsen, Angelika Mader, Peter Niebert, Barend van den Nieuwelaar, Frits Vaandrager and Marcel Verhoef, thank you! Furthermore, I want to thank the reading committee, Rajeev Alur, Joost-Pieter Katoen and Sergio Yovine, for their effort and for providing many useful comments that have improved this thesis.

The daily life on the 6th floor is very pleasant. I thank all people from the ITA and SOS groups for this nice environment. I especially thank Mirèse Willems, Maria van Kuppenveld and Desiree Hermans for their help with the many practical problems. I also thank Harco Kuppens for his help with the often occurring computer difficulties. Without his handcrafted backup utility that ran in parallel with the C&CZ backup (that cannot always be trusted as I experienced. . .), I would have had to retype this whole thesis during the last fall.

During my stay in the ITT group I have also participated in some teaching activities which I found very interesting and useful. Therefore I thank Hanno Wupper, Hans Meijer and Stijn Hoppenbrouwers for giving me the opportunity to gather some teaching experience in the *Beweren en Bewijzen* course.

In the curriculum vitae that is included at the end of this book, you can read that I started out studying chemistry. That did not work out very well and I decided to switch to computer science. This proved to be a very good choice and I thank my family for supporting it.

Femke, thank you for your love.

Contents

Chapter 1

Introduction

1.1 The Quality of Computer Systems

We are surrounded by computation devices. Our CD players, mobile phones, and cars all contain some kind of computation unit that controls (part of) the device and provides some highly specific functionality. Typically, such computation units are reactive and are embedded in a complex environment. The design of these *embedded systems* is not a trivial task, since (i) they perform complex interactions with the continuous environment, and (ii) they must meet high dependability requirements. The difficulty of this task has recently (once again) been demonstrated by a software problem that caused some Toyota Prius gas-electric hybrid cars to stall or shut down while driving at highway speeds[1]. Consequently, many techniques have been developed for the design of these systems that must ensure that they will satisfy their requirements. For instance, quality management (such as defined by the ISO 9000 standard; see `http://www.iso.org/`) and process assessment and improvement (such as provided by the Carnegie-Mellon Software Engineering Institutes Capability Maturity Model (CMM); see `http://www.sei.cmu.edu/`) are techniques that exist on a corporate level. These two topics are not discussed here since they fall outside the scope of this thesis. On the project level there exist techniques such as model-based development, formal analysis and testing to guarantee that systems satisfy their requirements. These are discussed below.

It is essential to handle the complexity of the construction of embedded systems. This can be done by using both different views on the system and different levels of detail (abstractions). *Model-based development* aims to be an incremental process which integrates different models on different levels of abstraction in a consistent way in order to improve the quality and efficiency of the development process (see, e.g., [95]). The methodology uses formal modeling languages (i.e., possibly graphical languages with a precisely defined syntax and semantics) to specify (parts of) systems[2]. An intrinsic merit of a formal language is that it forces one to specify the system in a precise and unambiguous way, which may reveal inconsistencies and gaps in the original informal description. For instance, it may reveal non-determinism within a component, which could be an error since it cannot be implemented, but which also could be a postponed design decision. Furthermore, a formal language can be used for analysis since it has mathematically defined semantics.

[1] `http://money.cnn.com/2005/05/16/Autos/prius_computer/index.htm`

[2] There are too many formal languages to enumerate here. For an overview, see the world wide web library on formal methods at `http://www.afm.sbu.ac.uk/`.

Two important formal analysis techniques, which may be applied stand-alone or during model-based development, are *theorem proving* and *model checking*. A theorem prover, such as ISABELLE [88] or PVS [89], supports the manual construction of proofs by checking the correctness of the proof steps, and provides some automation to the proof process. In general, theorem provers require a lot of human interaction to complete a proof. Model checking, on the other hand, is a methodology to automatically analyze problems that have been expressed in terms of transition systems [37]. This thesis concerns model checking, and the next sections offer a more detailed explanation of this topic. Both model checking and theorem proving generally analyze abstract models rather than the executable code that implements those models. Furthermore, the hardware on which the executable code runs could also contain errors. This is only one reason why an implementation of a formally verified abstract design still needs to be *tested*. Note that research is emerging that aims to model check real code; see for instance the TAXYS tool [96], the SLAM tool [11], the BLAST tool [63] and the BOGOR tool [46].

Testing is widely used to assess the requirements of realized (sub)systems and typically requires a lot of effort: sometimes up to 50% of the project resources is spent on it. In contrast to model checking and theorem proving, which are complete with respect to the model, testing is in general incomplete. It can thus only be used to find errors, and not to prove their absence. Another, albeit less fundamental, problem with testing is that the conventional testing methodology is not really suitable for concurrent systems, since it provides no real control of the interleaving of concurrent processes. Recently, *model-based testing* has received much interest [32]. This approach seems to fit seamlessly into the model-based development approach.

Testing and formal analysis techniques are complementary. Usually, formal analysis is applied to assess requirements that are relatively easy to define on high-level designs. One of the strengths of formal analysis is that it can deal systematically with concurrency. A verified design, however, does not guarantee a correct implementation. Therefore, testing still is needed to validate that the implementation satisfies the requirements.

1.2 Model Checking

As the name implies, model checking requires the construction of a model of the system under consideration. In general, a model is a (possibly) infinite discrete event dynamical system consisting of states and labeled transitions. A model usually is given implicitly by describing the system in some higher-level modeling language. Model checking also requires the definition of the specification of the system, which usually is a temporal logic formula such as “no bad situation is ever reachable”. A model checking tool can then compute whether the model satisfies its specification. Nowadays, model checking tools are available for many application areas, e.g., hardware systems [45, 83], finite-state distributed systems [65],

timed and hybrid systems [16, 62, 102, 107], and software [11, 46, 63, 96, 101].

The model checking methodology can be illustrated with help from the popular Sudoku puzzle (`http://en.wikipedia.org/wiki/Sudoku`). This Japanese puzzle consists of a grid of 9 by 9 cells that is sectioned into 9 blocks of 3 by 3 cells, and which is partially filled with seemingly random digits from 1 through 9. A *valid* Sudoku grid does not contain equal numbers (e.g., two times the number three) in a row, column or block. The goal is to complete all cells such that every row, every column and every block contains every digit from 1 through 9. A model of a Sudoku puzzle consists of the definition of the initial state of the puzzle and of the definition of the rules that are allowed to change the state of the puzzle. Clearly, the initial state is the partially filled grid. The transition relation can then be defined as follows: an empty cell can be filled with a digit, only if the result still is a valid Sudoku grid. Figure 1.1 shows a part of the dynamics of a given puzzle.

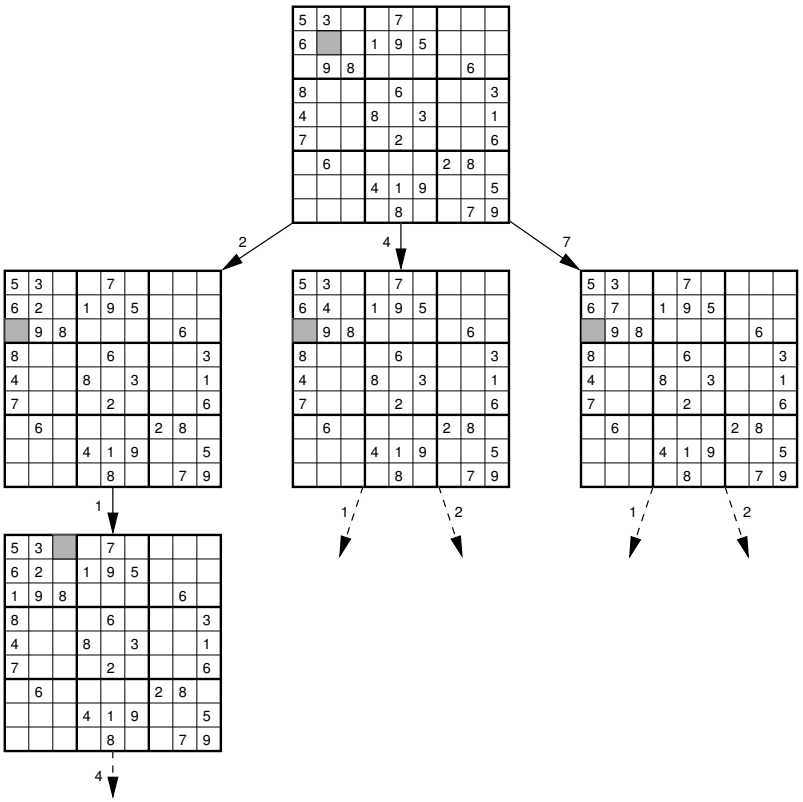

Figure 1.1: A very small part of the transition system that defines the dynamics of a given Sudoku puzzle. The uppermost state is the initial state. Note that only the transitions involving the marked cells are shown.

In order to find the solution to the puzzle, the model checker can be asked whether a state where each cell is filled by a digit can be reached. This is a typical

verification question. A typical optimization question is to ask the model checker for the shortest sequence of moves to the solution. The tool tries to answer these questions by a possibly exhaustive exploration of all reachable states and, if appropriate, it can supply a path in the transition system that explains the given answer.

Despite the fact that model checkers are relatively easy to use as compared to theorem proving, they are not applied on a large scale. Some reasons for this are mentioned below. Note that these reasons apply to general model checkers such as UPPAAL and SPIN, but not to specialized in-house industrial tools that incorporate model checking techniques.

1. *Scalability*. Model checkers must cope with the *state space explosion* problem, which is the problem of the exponential growth of the state space as models become larger. This growth often renders the mechanical verification of realistic systems practically impossible: there just is not enough time or memory available. For instance, the number of valid Sudoku solution grids for the standard 9 by 9 grid was calculated by Bertram Felgenhauer in 2005 to be 6,670,903,752,021,072,936,960, which is roughly the number of micrometers to the nearest star. The possible search space for a given Sudoku puzzle may thus be huge. Heuristics are often used to prune such huge search spaces. A very simple heuristic is good enough to solve many[3] Sudoku puzzles with the UPPAAL model checker (see `http://www.cs.ru.nl/M.Hendriks/sudoku.zip` for the model).

2. *Accessibility*. Building a good model is difficult because model checkers are mostly academic tools that lack extensive documentation and require a thorough knowledge of the underlying principles to build models that are suitable for analysis. Thus, in practice model checking tools are inaccessible to people with little or no background in formal verification.

3. *Convenience*. Model checkers usually are not a part of the development tool-chain with the result that there is little or no automation. Furthermore, many current tools and their input formalisms lack important features for convenient specifications in an industrial setting. As a result, modeling and analysis require a significant amount of time.

4. *Realizability*. It very often is unclear what the relation between the model and the reality is. One can verify a high-level design, but what does that say about the realization of that design?

At this point one might think that model checking tools are only suitable to solve small puzzles and to verify toy examples. This is definitely untrue. Many examples exist of non-trivial case studies in which model checking tools played an essential role. Because of the number and the variety of the available examples,

[3] All puzzles that were tried were solved within a few seconds, including some 17-hint Sudokus which are the most difficult Sudoku puzzles known.

it is not possible to give a short list of those that have had the greatest impact. Instead, the reader is referred to conferences where these case studies are regularly published, such as the conference series on Computer Aided Verification (CAV), and Tools and Algorithms for the Construction and Analysis of Systems (TACAS).

1.3 Overview of this Thesis

As indicated in the title, this thesis concerns the model checking of timed automata. The timed automaton framework is an automata based formalism that can capture quantitative timing aspects [7, 8]. Its modeling language extends finite state automata with real-valued clocks. For instance, a timer which generates a "tick" every 9 to 10 time units and which may fail at any moment can be modeled very naturally as a timed automaton: see Figure 1.2.

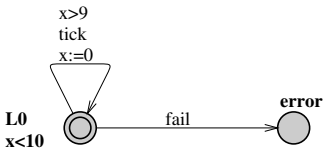

Figure 1.2: A timed automaton that models a timer which generates a *tick* every 9 to 10 time units, and which may *fail* at any moment.

This automaton has two *locations* namely *L0*, which is the *initial* location, and *error*. It also has one *clock*, namely x. Location *L0* is labeled with the *invariant* "$x < 10$" which specifies that the value of clock x must be smaller than 10 whenever location *L0* is active. The self-loop is labeled with the *clock guard* "$x > 9$", which is the enabling condition for the edge. Furthermore, it is labeled with an action *tick*, and with the *clock reset* "$x := 0$". The semantics of a timed automaton is defined by an infinite labeled transition system. The states in this system are tuples (l, ν), where l is the current location of the automaton, and ν is a function that maps the clock of the automaton to a non-negative real number. There are two types of transitions: (i) *delay* transitions that model the elapse of time, and (ii) *action* transitions that execute an edge of the automaton. For instance, the following is a possible path of the automaton shown in Figure 1.2:

$$(\mathit{L0}, x = 0) \rightarrow^{4.3} (\mathit{L0}, x = 4.3) \rightarrow^{5.1} (\mathit{L0}, x = 9.4) \rightarrow^{\mathit{tick}} (\mathit{L0}, x = 0) \rightarrow^{2.7}$$
$$(\mathit{L0}, x = 2.7) \rightarrow^{0.2} (\mathit{L0}, x = 2.9) \rightarrow^{\mathit{fail}} (\mathit{error}, x = 2.9) \rightarrow^{8.1} (\mathit{error}, x = 11)$$

Although the model of a timed automaton has an infinite state space, techniques have been developed to analyze it algorithmically. The basis for this analysis is the so-called *region automaton*, which is a finite abstraction that preserves *Timed Computation Tree Logic* (TCTL) [6]. Unfortunately, the clocks impose a significant burden: typically the region automaton is just too big to fit in the memory of a regular desktop computer. This problem is addressed in [64].

The *zone automaton*, which preserves only a subset of TCTL (most notably reachability) and is often usable in practice, is mostly implemented by model checking tools for timed automata such as UPPAAL [16].

Timed automata have proven to be a natural formalism for the specification and verification of real-time systems. During the last years it has been recognized that timed automata are also very suitable for the specification, verification and optimization of all kinds of resource allocation problems in which time plays an essential role, such as job-shop and task-graph scheduling problems [3, 2] and industrial scheduling problems [49] (also see Chapters 5 and 6.).

1.3.1 Aim of this Thesis

All work described within this thesis has been carried out in the context of the EU project IST-2001-35304 AMETIST [9]. The following excerpt from the final project report defines the aim of AMETIST: "Whereas timed automata and the tools for their analysis are widely accepted in academia and are being used at hundreds of universities and research laboratories around the world, they have yet to find their way into industry. The aim of AMETIST has been to advance and mature the related models, tools, and methods to allow this situation to change." The four drawbacks of model checking technology that have been mentioned above impair the use of timed automata technology in industry, and AMETIST aimed to (at least partially) eliminate them.

> The aim of this thesis is to develop and implement new techniques to alleviate the state space explosion problem for timed automata, and to demonstrate and evaluate the practical applicability of timed automata model checking tools.

The first goal coincides perfectly with the *scalability* issue that AMETIST aimed at, whereas the second goal contributes to research of the *accessibility* and *convenience* issues. The research methodology that has been applied in this thesis to reach these goals is practice-driven and centered around the UPPAAL tool. The basis consists of case-studies which have contributed to the existing body of experience and have shown the current capabilities and shortcomings of UPPAAL with respect to modeling and analysis issues. The attempts to analyze the case-studies with UPPAAL have suggested ways to improve (i) the tool and (ii) analysis techniques for timed automata in general. Consequently, implementation of these suggestions has resulted in a more mature tool, which closes the cycle.

1.3.2 Content of this Thesis

This thesis consists of six research papers, namely [59, 55, 57, 60, 15, 56], that are summarized below.

- Chapter 2: *Exact Acceleration of Real-Time Model Checking*. This chapter deals with the state space explosion that results when various system components have different time scales (for instance, a controller that is based on microseconds and its environment that is based on seconds). The main contribution is a theorem that alleviates this problem for a subclass of timed automata.

- Chapter 3: *Enhancing Uppaal by Exploiting Symmetry*. Symmetry reduction is a well-known technique to reduce the state space when multiple components that "behave the same" are present in the system. The contribution of this work is a soundness proof that transfers symmetry reduction from an untimed setting to a timed setting.

- Chapter 4: *Adding Symmetry Reduction to Uppaal*. In this chapter it is shown that the symbolic representation of the clocks does not complicate the symmetry reduction technique that has been proposed in Chapter 3. The technique has been implemented in a prototype of UPPAAL, and experiments show a exponential improvement in both time and space for some models.

- Chapter 5: *Model Checker Aided Design of a Controller for a Wafer Scanner*. This case study shows that model checking techniques can be used to solve a verification and optimization problem on different levels of abstraction within a single framework. This work is referred to in patent application ASML ref. P-1784.010, which shows its relevance for industry.

- Chapter 6: *Production Scheduling by Reachability Analysis*. Timed automata that are extended with cost functions are used in this case study to find schedules for lacquer production. It is shown that the model checker based approach can compete with a commercial value chain optimization tool for this particular case study.

- Chapter 7: *Model Checking the Time to Reach Agreement*. This chapter presents a typical verification problem that is very difficult for model checkers for various reasons and was considered far out of reach only three years ago.

The research for the exact acceleration technique (Chapter 2) was triggered by failed attempts to analyze the behavior of executable byte code for the LEGO RCX brick [54]. Similarly, the research for symmetry reduction (Chapters 3 and 4) was triggered by failed attempts to analyze interesting instances of the agreement algorithm that is presented and analyzed in Chapter 7. The fact that such instances could be verified three years later without the use of symmetry reduction shows the steady progress of the efficiency of UPPAAL. Furthermore, the AXXOM case study (Chapter 6) triggered a number of small but helpful enhancements of UPPAAL.

1.4 Conclusions

It has been known for many years that model checking techniques can contribute to the design and analysis of real systems. A lot of research concerning model checking has been, and still is, directed at the fundamental scalability problem. This has resulted in very powerful tools that enable experts in the field of formal verification to solve many interesting real-life case-studies fast and in a routine manner. The ASML case study (Chapter 5) and the agreement case study (Chapter 7) are relevant examples in this respect.

The fact that many techniques to scale model checking are often applied simultaneously raises a practical question of correctness. Apart from the issue that the actual implementation of the model checking algorithm should be correct, it must also be ascertained that all applied techniques are pairwise compatible. Furthermore, the fact that modeling languages evolve poses another problem. For instance, the symmetry reduction technique for timed automata that is presented in Chapters 3 and 4 has been based on a version of UPPAAL which did not yet contain the C-like language that the current development version contains and which the next major release will contain. How is symmetry reduction handled in this new language? Must all theoretical work be done again to ensure soundness?

Although at least two of the three case studies that are presented in this thesis have been solved quickly in a routine manner, the case studies also have shown that model checking is not yet push-button technology. Three prominent problems that remain (apart from the always existing scalability problem) are the following:

- Modeling the problem in a logical and straightforward manner may result in a model that is too detailed to analyze.
- It is often difficult and inconvenient to model search heuristics.
- It may be very difficult to model a specific part of the problem domain.

The first of the above problems is closely related with the scalability problem, and will, just like the scalability problem, always be present. An effective way to circumvent this problem is to construct abstractions that are just detailed enough to be useful. This, however, can be a highly non-trivial task, as is demonstrated by the agreement case study. The ASML case study provides a nice example of the use of different levels of abstraction: the verification problem is solved on an abstract model, and the optimization problem is solved on a concrete model. Furthermore, it is manually proven that the abstraction is sound. Ideally, of course, the process of making abstractions or refinements and proving them sound needs no (or just a little) human interaction.

The second problem typically applies to scheduling problems. Usually, the state space of these models is far too large to handle and heuristics are applied to guide the search for good schedules. The modeling of heuristics often is non-trivial, such as the *non-laziness* heuristic in the AXXOM case study (Chapter 6) and

the *start-up* heuristic in the ASML case study. This clearly poses an accessibility problem for the model checker based approach. Furthermore, modeling heuristics is not convenient since separating them cleanly from the core of the model often is not possible. This quickly results in many versions of a model, resulting in the obvious problems.

The third problem is is illustrated by the working hours of the personnel in the AXXOM case study. Modeling this constraint with timed automata does not feel natural and is very laborious. This is a typical accessibility problem: modeling the constraint is difficult and extra care has to be taken to avoid that a very inefficient model results. The conclusion therefore is that raw timed automata, just like any other low-level formalism, may just not be very suitable for these kind of constraints.

The second and third problem can be circumvented by high-level domain-specific languages that form a front-end for timed automata models. Current research investigates this approach for architectural design-space exploration [61]. An alternative is to provide a library of *problem templates* for specific application domains. A user can select a problem template that models a problem that is most similar to its own problem (including useful heuristics). Ideally, only small and easy to understand changes in the model are needed to transform it to a suitable model for the new problem. This approach may work for many cases. When it does not work, however, the need for detailed knowledge of timed automata rises again.

The AMETIST project has made huge progress in dealing with the fundamental scalability problem: the performance of UPPAAL has improved several orders of magnitude during this three-year project. Still, there are many existing techniques that could be added to UPPAAL to improve its performance even more, such as dead-variable reduction for integer variables [106] and clock optimization [40], slicing based on the verification property [29], the sweep-line method[4] [36], and alternative symbolic techniques such as presented in [103]. Furthermore, the problems of convenience and accessibility seem to become more prominent with the increase of the efficiency of model checking tools. One way to solve these is to provide high-level languages for specific application domains that translate to efficient low-level models that can be handled by existing tools. Another way is to build a "case study library" from which problems that have already been solved can be taken and adjusted to the existing problem.

[4] Work on an implementation of the sweep-line method in UPPAAL has recently been initiated.

Chapter 2

Exact Acceleration of Real-Time Model Checking

MARTIJN HENDRIKS

KIM LARSEN

Abstract. Different time scales do often occur in real-time systems, e.g., a polling real-time system samples the environment many times per second, whereas the environment may only change a few times per second. When these systems are modeled as (networks of) timed automata, the verification using symbolic model checking techniques can significantly be slowed down by unnecessary *fragmentation* of the symbolic state space. This paper introduces a syntactical adjustment to a subset of timed automata that addresses this fragmentation problem and that can speed-up forward symbolic reachability analysis in a significant way. We prove that this syntactical adjustment is exact w.r.t. reachability properties and that it indeed is effective. We illustrate our *exact acceleration* technique with run-time data obtained with the UPPAAL model checker. Moreover, we demonstrate that automated application of our exact acceleration technique can significantly speed-up the verification of the run-time behavior of LEGO Mindstorms programs.

2.1 Introduction

Model checking systems in which various components use very different time scales suffers from the *fragmentation problem*. This, for example, is often the case for models of reactive programs with their environment. This difference can give rise to an unnecessary fragmentation of the symbolic state space: busy waiting of one of the components in the model slices the time even when nothing interesting is happening. As a result, the time and memory consumption of the model checking process increases.

The fragmentation problem has first been encountered and described by Hune and Iversen et al during the verification of LEGO MINDSTORMS programs using UPPAAL [67, 70]. The symbolic state space is severely fragmented by the busy waiting behavior of the control program components. Other examples that may suffer from fragmentation include the aforementioned reactive programs, and polling real-time systems, e.g., programmable logic controllers [41]. We propose an acceleration technique that addresses the fragmentation problem for a subset of timed automata that contain special busy waiting cycles. Our technique consists of a syntactical adjustment that can easily be computed from the timed automaton itself.

This chapter is an improved version of [59].

We prove that the application of this syntactical adjustment is exact with respect to reachability properties and that it can effectively speed-up forward symbolic reachability analysis. As a result, our approach is readily applicable using the existing model checkers.

Related work. Our approach has been heavily inspired by Möller's "parking" approach to the sketched fragmentation problem, which is based on a syntactical adjustment that gives – in general – an over approximation of the state space [85]. In fact, we have taken a special case of "parking" and we have proved its exactness w.r.t. reachability. We think that both methods show promises for handling the fragmentation problem. Closely related work has been done in the field of symbolic verification of systems that are modeled by a discrete control graph with unbounded integer variables [25]. Static analysis of the control graph is used to detect interesting cycles, of which the result of iterated execution can be computed by a single meta transition. These meta transitions are then added to the system and favored by the state space exploration algorithm, resulting in faster exploration of the state space. Symbolic techniques using *queue-content decision diagrams*, or QDDs, for the analysis of communication protocols that are modeled by finite-state machines that communicate through unbounded FIFO-queues, also use meta transitions to accelerate the exploration of the state space [23, 24]. Special cycles in the control-graph, e.g., the repeated receiving of messages from a channel, are associated with meta transitions that compute all states that are reachable by the iterated execution of the cycle. In these approaches only a limited class of cycles in the control graph can be accelerated due to the expressibility of QDDs. To overcome this problem, *constrained* QDDs have been introduced, that allow the acceleration of any cycle in a control graph [27]. Recently, acceleration techniques have been proposed in the setting of parameterized model checking [4, 93]. Again, the idea is to compute the effect of an unbounded number of actions to accelerate the forward exploration process.

Outline. In Section 2.2 we briefly summarize the syntax and semantics of timed automata. Section 2.3 explains the basic definitions and lemmas and in Section 2.4 the main theorems are presented. Finally, Section 2.5 gives an application of exact acceleration, and Section 2.6 draws some conclusions.

2.2 Timed Automata

The timed automata framework which has been introduced by Alur and Dill [5, 8] is a formalism for specifying dense real-time systems. The basic definitions concerning timed automata from [5] are reused here. In order to define finite automata that use real valued clocks, first the set of clock constraints over a set of clock variables is defined. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints is inductively defined by the rules (1) $x \sim c \in \Phi(X)$ and (2) if $\phi_1, \phi_2 \in \Phi(X)$, then $\phi_1 \wedge \phi_2 \in \Phi(X)$, where $x \in X$, $c \in \mathbb{N} \cup \{\infty\}$ and $\sim \in \{<, \leq, =, \geq, >\}$. The symbol ∞ is allowed in clock constraints because it makes the definitions below

more elegantly. We define $c \leq \infty$ and $c + \infty = \infty$ for all $c \in \mathbb{R} \cup \{\infty\}$. A *clock interpretation* ν for a set of clocks X is a mapping from X to $\mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of positive real numbers including zero. A clock interpretation ν for X satisfies a clock constraint ϕ over X, denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν. Thus, the constant *true* can be defined by the constraints $x \geq 0$ and $x \leq \infty$ where $x \in X$. The notation $\nu + \delta$, where $\delta \in \mathbb{R}^+$, is used for the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. The notation $\nu[Y := 0]$, where $Y \subseteq X$, is used for the clock interpretation which assigns 0 to each $x \in Y$ and leaves the other clocks unchanged w.r.t. ν. We let $\Gamma(X)$ denote the set of all clock interpretations for X.

Definition 2.1 (Timed Automaton) *A* timed automaton *is a tuple* $(L, l^0, \Sigma, X, I, E)$*, where* L *is a finite set of locations,* $l^0 \in L$ *is the initial location,* Σ *is a finite set of labels,* X *is a finite set of clocks,* $I : L \to \Phi(X)$ *labels each location with some clock constraint, and* $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ *is a finite set of edges.*

An edge $e = (l, a, \phi, \lambda, l')$ represents a transition from location l to location l' on the symbol a. The clock constraint ϕ specifies when the edge is enabled and the set $\lambda \subseteq X$ gives the clocks to be reset with this edge. The source of e is l, and is denoted by *src*(e). The destination of e is l', and is denoted by *dst*(e).

The semantics of a timed automaton $(L, l^0, \Sigma, X, I, E)$ is defined by associating a transition system $(S, s^0, \to)$ with it. The set of states S consists of all pairs (l, ν), where $l \in L$ and $\nu \in \Gamma(X)$ such that ν satisfies $I(l)$. The initial state s^0 is the state (l^0, ν_{init}), where $\nu_{init}(x) = 0$ for all $x \in X$. We assume that $\nu_{init} \models I(l_0)$. There are two types of transitions in the transition system:

1. Let $(l, \nu), (l', \nu') \in S$ and let $\delta \in \mathbb{R}^+$. If $\nu' = \nu + \delta$, $l' = l$, and $\nu + \delta' \models I(l)$ for all $0 \leq \delta' \leq \delta$, then $((l, \nu), (l', \nu')) \in \to$.

2. Let $(l, \nu), (l', \nu') \in S$ and let $e = (l, a, \phi, \lambda, l') \in E$. If $\nu \models \phi$ and $\nu' = \nu[\lambda := 0]$, then $((l, \nu), (l', \nu')) \in \to$.

The first transition is a *δ-delay transition* and is abbreviated by $(l, \nu) \to^{\delta} (l', \nu')$. The second transition is an *e-action transition*, abbreviated by $(l, \nu) \to^{e} (l', \nu')$. Due to the fact the clock constraints are conjunctions of lower and upper bounds on clocks, the following lemma can be proved.

Lemma 2.2 (Convexity) *If* $(l, \nu), (l, \nu') \in S$ *and* $\nu' = \nu + \delta$ *for some* $\delta \in \mathbb{R}^+$*, then* $(l, \nu) \to^{\delta} (l, \nu')$.

PROOF. We must prove that $\nu + \delta' \models I(l)$ for all $0 \leq \delta' \leq \delta$. This can be done by proving that for all $\phi \in \Phi(X)$ holds that if $\nu' = \nu + \delta$, $\nu \models \phi$ and $\nu' \models \phi$, then $\nu + \delta' \models \phi$ for all $0 \leq \delta' \leq \delta$ by induction on the syntax of ϕ, which is straightforward. ■

Note that this lemma cannot be proved if disjunctions are allowed in clock constraints; a constraint like $x < 5 \vee x \geq 7$ can easily be used for a counter example.

Definition 2.3 (Path) *Let $\mathcal{M} = (L, l^0, \Sigma, X, I, E)$ be a timed automaton. A finite or infinite sequence $(l_0, \nu_0), (l_1, \nu_1), ...$ is an (l, ν)-*path *of $\mathcal{M}$ if and only if $l_0 = l$ and $\nu_0 = \nu$, and for all $i > 0$ holds $((l_{i-1}, \nu_{i-1}), (l_i, \nu_i))$ is an e-action transition for some $e \in E$ or a δ-delay transition for some $\delta \in \mathbb{R}^+$.*

Any state on a (l^0, ν_{init})-path is a reachable state. In order to characterize reachable sets of states we define *state properties* as sets of states of a timed automaton.

Definition 2.4 (Reachability and Invariance) *Let $\mathcal{M}$ be a timed automaton, let s be a state of $\mathcal{M}$, and let ϕ be a state property of $\mathcal{M}$. A state that satisfies ϕ is* reachable *from s in $\mathcal{M}$, denoted by $(\mathcal{M}, s) \models \mathbf{EF}(\phi)$ if and only if an s-path $s_0, s_1, \ldots, s_n$ of $\mathcal{M}$ exists such that $s_n \in \phi$. A state that satisfies ϕ is* invariant *from s in $\mathcal{M}$, denoted by $(\mathcal{M}, s) \models \mathbf{AG}(\phi)$ if and only if for every finite or infinite s-path $s_0, s_1, \ldots$ of $\mathcal{M}$ holds that $s_i \in \phi$ for all $i \geq 0$.*

Let $\mathcal{M}$ be a timed automaton and let π be an s-path of $\mathcal{M}$. We say that π is *compressed* if and only if it starts with a delay transition, any delay transition is either followed by an action transition or it is the last transition, and any action transition is either followed by a delay transition or it is the last transition. Every finite path can be converted to a compressed path as follows: First, insert a 0-delay transition before every action transition, and second, replace n consecutive delay transitions with delays $\delta_1, \ldots, \delta_n$ by a $\sum_{i=1}^{n} \delta_i$-delay transition. We let *compress*(π) denote the compressed version of a path π.

2.3 Predictable Delays of Edge Sequences

Consider some timed automaton $\mathcal{M}$ and let $\sigma = e_1, e_2, \ldots, e_n$ be a sequence of edges of $\mathcal{M}$. We define *src*(σ) = *src*(e_1). We say that σ is *consecutive* if and only if *dst*(e_i) = *src*(e_{i+1}) for all $1 \leq i < n$. Furthermore, σ is *cyclic* if and only if it is consecutive and *dst*(e_n) = *src*(e_1). An *execution of* σ is a path, say π, that starts in $src(\sigma)$ and that exactly takes the edges in σ and ends with a e_n-action transition, i.e., *compress*(π) has the form

$$(l_1, \nu_1) \rightarrow^{\delta_1} (l_1, \nu_1') \rightarrow^{e_1} (l_2, \nu_2) \rightarrow^{\delta_2} \ldots \rightarrow^{\delta_n} (l_n, \nu_n') \rightarrow^{e_n} (l_{n+1}, \nu_{n+1})$$

The accumulated delay of π equals $\sum_{i=1}^{n} \delta_i$, and is denoted by *delay*(π). Finally, n repetitions of a sequence σ, denoted by σ^n, is defined as the sequence $\sigma_1, \ldots, \sigma_n$ where $\sigma_i = \sigma$ for all $1 \leq i \leq n$.

The next definition associates a *delay interval* with a sequence of edges if every execution of that sequence has a delay in that interval, *and* for every number δ in

the interval it is always possible to execute the sequence from a special set of states such that the delay equals δ. The main idea behind this definition is that it enables the exact computation of the result of the iterated execution of a cycle with a delay interval. This result can then be used for the exploration of the state space.

Definition 2.5 (Delay Interval) *Let σ be a consecutive edge sequence, and let y be a clock. An interval $[a, b]$, where $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$[1], is a* y-delay interval *of σ, if and only if*

1. *for any execution π of σ starting in a state (l, ν) such that $\nu(y) = 0$ holds that delay$(\pi) \in [a, b]$, and*
2. *for all $\delta \in [a, b]$ holds that in any state $(src(e_1), \nu)$ such that $\nu(y) = 0$ starts an execution π of σ such that delay$(\pi) = \delta$.*

Since a delay interval is unique, we speak of *the* delay interval of an edge sequence. The next definition is a syntactic property of an edge sequence that implies the previous semantic property.

Definition 2.6 (Predictable Delay) *Let $\sigma = e_1, \ldots, e_n$ be a consecutive edge sequence of a timed automaton $(L, l^0, \Sigma, X, I, E)$, and let $e_i = (l_i, a_i, \phi_i, \lambda_i, l_{i+1})$. Then σ has a* predictable delay for clock y, *if and only if*[2]*,*

1. *for all $1 \leq i \leq n + 1$ some $d_i \in \mathbb{N} \cup \{\infty\}$ exists such that $I(l_i) \equiv y \leq d_i$,*
2. *for all $1 \leq i \leq n$ either $\phi_i \equiv y \geq c_i$ or $\phi_i \equiv y = c_i$ for some $c_i \in \mathbb{N}$,*
3. *$c_i \leq d_i$ for all $1 \leq i \leq n$, and $c_i \leq d_{i+1}$ and $c_i \leq c_{i+1}$ for all $1 \leq i < n$,*
4. *$\lambda_n \equiv \{y\}$ and $\lambda_i \equiv \emptyset$ for all $1 \leq i < n$.*

Note that the third requirement is not really restrictive, since it states a "sanity" rule: violation of it introduces redundancy, deadlocks or may even have the result that the edge sequence has no executions. We claim that the specification of components that have an approximately known delay in their computation, e.g., the cycle of a programmable logic controller, often is of the form as defined above.

Lemma 2.7 *If an edge sequence has a predictable delay for y (with the naming of variables as in Definition 2.6), then it has a y-delay interval that equals $[c_n, c_n]$ if $\phi_n = y = c_n$, and $[c_n, d_n]$ otherwise.*

[1] The unconventional notation $\delta \in [a, \infty]$ means that $\delta \geq a$.

[2] The non-strictness constraints of the first and second item are present to simplify the definitions and proofs in the sense that less cases need to be considered. We claim, however, that they can be dropped resulting in only very minor changes to the theorems.

PROOF. Assume that the edge sequence $\sigma = e_1, e_2, \ldots, e_n$ has a predictable delay for y. We use the naming of variables as introduced in Definition 2.6. We first prove the first part of Definition 2.5. To this end, consider the compressed form of an execution of σ that starts in state (l_1, ν_1) such that $\nu_1(y) = 0$:

$$\begin{aligned}\pi = \quad & (l_1, \nu_1) \rightarrow^{\delta_1} (l_1, \nu_1') \rightarrow^{e_1} (l_2, \nu_2) \rightarrow^{\delta_2} \cdots \rightarrow^{e_{n-1}} \\ & (l_n, \nu_n) \rightarrow^{\delta_n} (l_n, \nu_n') \rightarrow^{e_n} (l_{n+1}, \nu_{n+1})\end{aligned}$$

First, note that *delay*(π) equals $\sum_{i=1}^{n} \delta_i$ by definition, which is equal to $\nu_n'(y)$ since $\nu_1(y) = 0$ and y is only reset on e_n by Definition 2.6. Now assume that ϕ_n has the form $y = c_n$. Then clearly, $\nu_n'(y) = c_n$, since otherwise e_n is not enabled. Therefore, *delay*$(\pi) = c_n \in [c_n, c_n]$. For the second case, assume that ϕ_n has the form $y \geq c_n$. Clearly, $\nu_n'(y) \geq c_n$, since otherwise e_n is not enabled. Hence, *delay*$(\pi) \geq c_n$. Furthermore, $\nu_n' \models I(l_n)$, since we assumed that (l_n, ν_n') is a state. Thus, $\nu_n' \models y \leq d_n$ by Definition 2.6, and therefore $\nu_n'(y) \leq d_n$. Hence *delay*$(\pi) \leq d_n$.

Next, we prove part 2 of Definition 2.5. Depending on the form of ϕ_n, let $\delta \in [c_n, c_n]$, or let $\delta \in [c_n, d_n]$. Let (l_1, ν_1) be a state such that $l_1 = \mathit{src}(e_1)$ and $\nu_1(y) = 0$. Consider the following sequence of pairs of locations and clock valuations:

$$\pi = (l_1, \nu_1), (l_1, \nu_1'), (l_2, \nu_2), (l_2', \nu_2'), \ldots, (l_n, \nu_n), (l_n, \nu_n'), (l_{n+1}, \nu_{n+1})$$

where the clock valuations are defined as follows:

$$\begin{array}{lcl} \nu_i' & = & \nu_1 + c_i \text{ for all } 1 \leq i < n \\ \nu_i & = & \nu_1 + c_{i-1} \text{ for all } 2 \leq i \leq n \\ \nu_n' & = & \nu_1 + \delta \\ \nu_{n+1} & = & (\nu_1 + \delta)[\{y\} := 0] \end{array}$$

Informally, the delay in every location – except for l_n – is kept as small as possible to enable the guard of the outgoing edge. In location l_n the total delay then is increased to match δ. This can all be done due to the requirements stated in Definition 2.6. A more formal proof is straightforward but tedious and is therefore omitted. ■

The proof of the previous lemma enables us to use the syntax of a timed automaton to compute the delay interval of an edge sequence with a predictable delay. Consider, for instance, the timed automaton depicted in Figure 2.1. It is an abstract model of the parallel composition of some control program and an environment which is modeled by clock z and the (very large) constant L. It has two consecutive edge sequences that have a predictable delay for clock y, namely $(\mathit{L1}, \tau, \mathit{true}, \{y\}, \mathit{L2})$ and $(\mathit{L2}, \tau, y \geq 1, \emptyset, \mathit{L3}), (\mathit{L3}, \tau, y \geq 3, \{y\}, \mathit{L1})$ (τ is the "empty" label and is not depicted). The delay interval for the edge from *L1* to *L2* equals $[0, 2]$ and the delay interval for the edges between the locations *L2*, *L3*, *L1* equals $[3, 5]$.

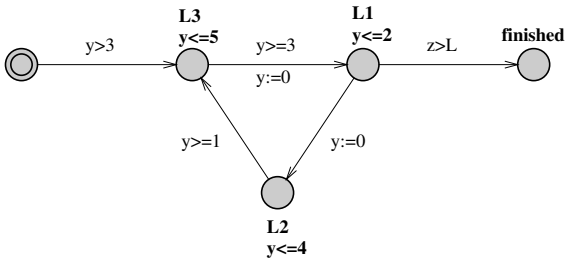

Figure 2.1: Timed automaton P.

It would be convenient to be able to sum the y-delay intervals of consecutive edge sequences to obtain the y-delay interval of the total edge sequence. A lemma is proved below that formalizes this. First, however, an intermediate result is proved.

Lemma 2.8 *Let $a_1, a_2 \in \mathbb{N}$, let $b_1, b_2 \in \mathbb{N} \cup \{\infty\}$, let $a_1 \leq b_1$ and let $a_2 \leq b_2$. If $\delta \in [a_1 + a_2, b_1 + b_2]$, then a $\delta_1 \in [a_1, b_1]$ and $\delta_2 \in [a_2, b_2]$ exist such that $\delta = \delta_1 + \delta_2$.*

PROOF. We distinguish three cases.

1. $b_1 = b_2 = \infty$. We let $\delta_1 = a_1$ (thus $\delta_1 \in [a_1, b_1]$) and $\delta_2 = \delta - \delta_1$ (thus $\delta = \delta_1 + \delta_2$). Since $\delta \geq a_1 + a_2$ it holds that $\delta_2 \geq a_2$. Thus $\delta_2 \in [a_2, b_2]$ since $b_2 = \infty$.

2. $b_1 = \infty$ and $b_2 \neq \infty$. (The case $b_1 \neq \infty$ and $b_2 = \infty$ obviously is similar.) We let $\delta_2 = a_2$ and $\delta_1 = \delta - \delta_2$. The proof is similar to the previous case.

3. $b_2 \neq \infty$ and $b_2 \neq \infty$. Without loss of generality we may assume that $a_1 + b_2 \leq b_1 + a_2$. Now let $\delta \in [a_1 + a_2, b_1 + b_2]$. We distinguish three cases:

   (a) $\delta \in [a_1 + a_2, a_1 + b_2]$. Then $\delta_1 = a_1$ and $\delta_2 = \delta - a_1$. Clearly, $\delta_1 \in [a_1, b_1]$ and $\delta_2 \in [a_2, b_2]$.

   (b) $\delta \in [a_1 + b_2, b_1 + a_2]$. Then $\delta_1 = \delta - b_2$ and $\delta_2 = b_2$. Clearly, $\delta_2 \in [a_2, b_2]$. It also clearly holds that $\delta_1 \geq a_1$. Furthermore, $\delta_1 \leq b_1 + a_2 - b_2$. Since $a_2 \leq b_2$, we can conclude that $\delta_1 \leq b_1$.

   (c) $\delta \in [b_1 + a_2, b_1 + b_2]$. Then $\delta_1 = b_1$ and $\delta_2 = \delta - b_1$. Clearly, $\delta_1 \in [a_1, b_1]$ and $\delta_2 \in [a_2, b_2]$.

■

Lemma 2.9 *Let σ be a consecutive edge sequence, and suppose that it can be written as $\sigma_1, \ldots, \sigma_n$ such that σ_i has a y-delay interval of $[a_i, b_i]$ where $a_i \in \mathbb{N}$ and $b_i \in \mathbb{N} \cup \{\infty\}$, and y is reset on every last edge of σ_i. Then σ has a y-delay interval of $[\sum_{i=1}^{n} a_i, \sum_{i=1}^{n} b_i]$.*

PROOF. The lemma can be proved by induction on n. The base case in which σ can be written as σ_1 is trivial. Now assume that the lemma holds for $n = k$. Consider the case for $n = k + 1$, i.e, $\sigma = \sigma_1, \ldots, \sigma_k, \sigma_{k+1}$. We first prove the first part of Definition 2.5. Therefore, consider an execution of σ that starts in a state (l_1, ν_1) such that $\nu_1(y) = 0$ (and let e_k be the last edge of σ_k and let e_{k+1} be the last edge of σ_{k+1}):

$$\pi = \underbrace{(l_1, \nu_1) \rightarrow \cdots \rightarrow^{e_k} (l_p, \nu_p)}_{\text{Execution of } \sigma_1, \ldots, \sigma_k} \rightarrow \cdots \rightarrow^{e_{k+1}} (l_q, \nu_q)$$

By the induction hypothesis we know that the delay up to state (l_p, ν_p), denoted by δ_1, is an element of $[\sum_{i=1}^{k} a_i, \sum_{i=1}^{k} b_i]$. Furthermore, clock y is reset on edge e_k by assumption. Therefore, $\nu_p(y) = 0$, and we can use Definition 2.5 to conclude that the delay δ_2 of the execution of σ_{k+1}, i.e., the path $(l_p, \nu_p) \rightarrow \cdots \rightarrow^{e_{k+1}} (l_q, \nu_q)$ takes a value in $[a_{k+1}, b_{k+1}]$. Clearly, $\delta_1 + \delta_2 \in [a_{k+1} + \sum_{i=1}^{k} a_i, b_{k+1} + \sum_{i=1}^{k} b_i]$.

Next, we prove the second part of Definition 2.5. Let (l_1, ν_1) be a state such that $l_1 = \mathit{src}(\sigma)$ and $\nu_1(y) = 0$, and let $\delta \in [\sum_{i=1}^{k+1} a_i, \sum_{i=1}^{k+1} b_i]$. By Lemma 2.8, we can write δ as $\delta_1 + \delta_2$, where $\delta_1 \in [\sum_{i=1}^{k} a_i, \sum_{i=1}^{k} b_i]$ and $\delta_2 \in [a_{k+1}, b_{k+1}]$. Using the induction hypothesis and Definition 2.5, an execution of $\sigma_1, \ldots, \sigma_k$ exists that starts in (l_1, ν_1), say, $(l_1, \nu_1) \rightarrow \cdots \rightarrow^{e_k} (l_p, \nu_p)$ with delay δ_1. By assumption, σ is consecutive which means that $l_p = \mathit{src}(\sigma_{k+1})$. Furthermore, y is reset on edge e_k by assumption, and therefore $\nu_p(y) = 0$. We can thus use Definition 2.5 to extend this path with an execution of σ_{k+1} that has delay δ_2. Thus, the resulting path is an execution of σ with delay $\delta_1 + \delta_2 = \delta$. ■

2.4 Acceleration of Timed Automata

Cycles can also have a delay interval, and when they do, the result of an arbitrary number of executions of the cycle can be computed in one step.

Definition 2.10 (Acceleratable Cycle) *An edge sequence σ is a* y-acceleratable cycle *if and only if it is cyclic, it can be written as $\sigma_1, \ldots, \sigma_n$ such that each σ_i has a predictable delay for y, and y is reset on every incoming edge of the first location of σ.*

Note that every y-acceleratable cycle has a y-delay interval that can easily be computed from the syntax (by Definition 2.6, Lemma 2.7, and Lemma 2.9). Con-

sider, for instance, the automaton in Figure 2.1. The cycle that starts in location *L1* is a y-acceleratable cycle and has a y-delay interval of $[3, 7]$.

The following definition appends an additional cycle to timed automata. We will see that a well-chosen additional cycle may be used to compute the effect of the iterated execution of an acceleratable cycle.

Definition 2.11 (Extension) *Let $\mathcal{M} = (L, l^0, \Sigma, X, I, E)$ be a timed automaton, let $l \in L$, let $b \in \mathbb{N}$ and let $x \in X$. The timed automaton $\mathcal{A}(\mathcal{M}, l, x, b) = (L \cup \{l'\}, l^0, \Sigma, X, I', E \cup \{e_1, e_2\})$ (assume that $l' \notin L$), where $I' = I \cup \{(l', true)\}$, $e_1 = (l, \tau, x \geq 0, \{x\}, l')$, and $e_2 = (l', \tau, x \geq b, \{x\}, l)$ is an* extension of $\mathcal{M}$.

This definition adds an extra cycle, the *additional cycle*, consisting of two edges to the automaton. Note that if l is on an y-acceleratable cycle of $\mathcal{M}$, then the additional cycle of the extension $\mathcal{A}(\mathcal{M}, l, y, b)$ has a y-delay interval that equals $[b, \infty]$. For instance, Figure 2.2 depicts a possible extension of location *L1* in which $x = y$ and $b = 3$ of the automaton in Figure 2.1. Clearly, the extension has a y-delay interval that equals $[3, \infty]$.

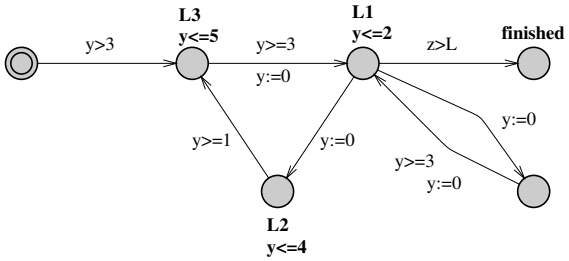

Figure 2.2: Timed automaton P_A: an extended version of P.

The extended automaton is only interesting if we can use it to model check properties of the original automaton. Next, we prove that some extensions are exact w.r.t. reachability properties. To this end, we first need an auxiliary lemma.

Lemma 2.12 *For all $\Delta \in \mathbb{R}$ and $a, b \in \mathbb{N}$ such that $a < b$ holds that if $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$, then an $n \in \mathbb{N}$ exists such that $\Delta \in [na, nb]$.*

PROOF. We consider two cases:

1. $a = 0$. We have that $\Delta \geq 0$ and we must find an $n \in \mathbb{N}$ such that $\Delta \in [0, nb]$. Since we assumed that $b > a$ we know that $b \geq 1$. The choice $n = \lceil \Delta \rceil$ proves this case.

2. $a > 0$. First, note that from $n \geq \frac{a}{b-a}$ easily follows that $(n+1)a \leq nb$ for all $a, b, n \in \mathbb{N}$. Next, we prove this case. To this end, let $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$. Clearly, some $n \in \mathbb{N}$ exists such that $na \leq \Delta < (n+1)a$. Since $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$, we can conclude that $n \geq \lceil \frac{a}{b-a} \rceil$. Thus clearly $n \geq \frac{a}{b-a}$, and hence $(n+1)a \leq nb$. Since $na \leq \Delta < (n+1)a$ we conclude that $\Delta \in [na, nb]$.

■

Theorem 2.13 (Exactness) *Let $\mathcal{M}$ be a timed automaton, let s^0 be the initial state of $\mathcal{M}$, let σ be a y-acceleratable cycle of $\mathcal{M}$ with a y-delay interval of $[a, b]$ such that $a < b$, and let ϕ be a state property of $\mathcal{M}$[3]. Consider the extension $\mathcal{M}' = \mathcal{A}(\mathcal{M}, src(\sigma), y, \lceil \frac{a}{b-a} \rceil \cdot a)$. Then*

$$(\mathcal{M}, s^0) \models \mathbf{EF}(\phi) \iff (\mathcal{M}', s^0) \models \mathbf{EF}(\phi)$$

PROOF. The $\Longrightarrow$ implication is trivial, since any path of $\mathcal{M}$ clearly also is a path of $\mathcal{M}'$. Now consider the converse implication. Let $src(\sigma) = l$. We prove that for any path $\pi = (l^0, \nu_{init}), \ldots, (l_f, \nu_f)$ in the extension such that $l_f \neq l'$, a path $\pi' = (l^0, \nu_{init}), \ldots, (l_f, \nu_f)$ in $\mathcal{M}$ exists. The proof is by induction on the number of occurrences of e_1-action transitions in π (see Definition 2.11). The case for 0 of those transitions is trivial, since then clearly also no e_2-action transitions occur in π, and as a result, π is a path in $\mathcal{M}$. Now assume that it holds for any path with m occurrences, and assume that π has $m + 1$ occurrences of e_1-action transitions. Then we can compress π and split π as follows:

$$\underbrace{(l^0, \nu_{init}) \cdots (l, \nu)}_{m \text{ occurrences}} \rightarrow^{\delta_1} (l, \nu') \rightarrow^{e_1} (l', \nu'') \rightarrow^{\delta_2} (l', \nu''') \rightarrow^{e_2} \underbrace{(l, \nu'''') \cdots (l_f, \nu_f)}_{0 \text{ occurrences}}$$

By the induction hypothesis, (l, ν) is also reachable in $\mathcal{M}$ from the initial state. Note that either $(l, \nu) = (l^0, \nu_{init})$ or an action transition leads to (l, ν). Since y is reset on every incoming edge of l (by Definition 2.10) and $\nu_{init}(y) = 0$ we can conclude that $\nu(y) = 0$. It also holds that $\nu''''(y) = 0$, since y is reset on e_2 by assumption. Furthermore, we know that no other clock than y is reset in the additional cycle. Therefore, $\nu''''(x) = \nu(x) + \Delta$ for all clocks $x \neq y$ and for some $\Delta \in \mathbb{R}^+$. By the assumptions from the theorem and Definition 2.11 we know that the additional cycle has a y-delay interval that equals $[\lceil \frac{a}{b-a} \rceil \cdot a, \infty]$. Thus, $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$ and we can use Lemma 2.12 to conclude that an n exists such that $\Delta \in [na, nb]$. Now consider the edge sequence σ^n in $\mathcal{M}$ (which equals n executions of the acceleratable cycle). Since σ is a cycle that that starts (and ends) in l, this edge sequence is consecutive. Furthermore, σ has a y-delay interval of $[a, b]$ by assumption, and y is reset on the last edge of σ by Definition 2.10. Therefore, we can use Lemma 2.9 to conclude that σ^n has a y-delay interval of $[na, nb]$. Hence, by Definition 2.5 we can indeed reach (l, ν'''') from (l, ν) in $\mathcal{M}$ by executing σ^n. Since the path from (l, ν'''') to (l_f, ν_f) in π does only use edges that are also present in $\mathcal{M}$, we conclude that a (l^0, ν_{init})-path that ends in (l_f, ν_f) also exists in $\mathcal{M}$. ■

[3]Remember that a state property is a set of states. Therefore, a state property of $\mathcal{M}$ is a state property of any extension of $\mathcal{M}$ (see Definition 2.11).

The extension P_A as depicted in Figure 2.2 satisfies the preconditions of the previous theorem. Therefore, this automaton can be used to model check reachability properties of the unaccelerated automaton P, which has been depicted in Figure 2.1. It may seem that the addition of the additional location cannot be a good idea since it only increases the state space. The next theorem, however, is the core of the explanation that it actually is a good idea when the difference in time scales is large.

Theorem 2.14 (Effectiveness) *Let $\mathcal{M}$ be a timed automaton with a acceleratable cycle σ that has a y-delay interval of $[a, b]$. Consider an extension $\mathcal{M}' = \mathcal{A}(\mathcal{M}, src(\sigma), y, \lceil \frac{a}{b-a} \rceil \cdot a)$ If $(l_1, \nu_1), \ldots, (l_1, \nu_1')$ is an execution of σ^n such that $n \geq \lceil \frac{a}{b-a} \rceil$ and $\nu_1(y) = 0$, then an execution $(l_1, \nu_1), \ldots, (l_1, \nu_1')$ of the additional cycle exists.*

PROOF. Note that the additional cycle has a y-delay interval of $[\lceil \frac{a}{b-a} \rceil \cdot a, \infty]$. Clearly, σ^n is consecutive since σ is a cycle. Furthermore, σ has a y-delay interval of $[a, b]$ by assumption, and y is reset on the last edge of σ by Definition 2.10. Therefore, we can use Lemma 2.9 to conclude that σ^n has a y-delay interval of $[na, nb]$.

Now consider an execution of σ^n, say $(l_1, \nu_1), \ldots, (l_1, \nu_1')$, such that $\nu_1(y) = 0$. Then $\nu_1'(y) = 0$ since y is reset on the last edge of σ by Definition 2.10 and Definition 2.6. Furthermore, by the same definitions we know that no other clock than y is reset in σ. Therefore, $\nu_1'(x) = \nu_1(x) + \Delta$ for all clocks $x \neq y$. Since σ^n has a y-delay interval of $[na, nb]$ and $n \geq \lceil \frac{a}{b-a} \rceil$, we know that the execution takes at least $\lceil \frac{a}{b-a} \rceil \cdot a$ time, i.e., $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$.

Using the assumption on the delay interval of the additional cycle and Definition 2.5 we know that for any $\delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$ an execution of the additional cycle starts in (l_1, ν_1) that has delay δ. By definition, such an execution ends in a state (l_1, ν_m) such that $\nu_m(y) = 0$ (since y is reset on the last edge of the additional cycle by Definition 2.11) and $\nu_m(x) = \nu_1(x) + \delta$ (since only clock y is reset in the additional cycle by assumption). Since $\Delta \geq \lceil \frac{a}{b-a} \rceil \cdot a$, we can choose $\delta = \Delta$ with the result that $\nu_m = \nu_1'$. Hence, the desired execution $(l_1, \nu_1), \ldots, (l_1, \nu_1')$ of the additional cycle exists. ■

Consider automaton P in Figure 2.1 and its accelerated version P_A in Figure 2.2. We want to know whether location *finished* is reachable. As mentioned before, model checkers for timed automata use a finite abstraction of the transition system to enforce decidability. States in the zone abstraction, which is used by the UPPAAL model checker, are tuples (l, Z), called *symbolic states*, where l is a location and Z is a zone, which is a set of clock valuations. For instance, when location *L1* is reached for the n-th time (the acceleratable cycle then is executed $n - 1$ times), then the zone of that state can be described by: $y \in [0, 2] \wedge z \in (3n, 7n] \wedge z - y \in (3n, 7n - 2]$. Figure 2.3 shows the zones that belong to states that are reached when location *L1* is visited for the first, second and third time. This clearly shows

the fragmentation problem. First, it takes many executions (and therefore a lot of computation time) to get to a state such that the transition to *finished* is enabled (since we assumed that the constant L is very large), and second, all these "small" zones are individually stored, which is not necessary since the union of a large subset of them is convex and can therefore be described by a single zone[4].

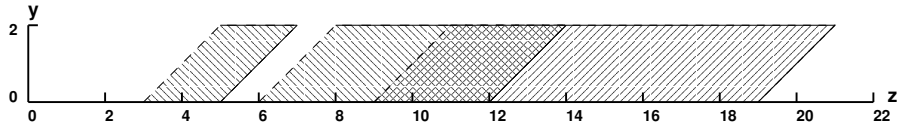

Figure 2.3: Three zones (the rightmost two overlap) that belong to states that are reached when location *L1* is visited for the first, second and third time.

Figure 2.4 shows the zone of that is reached after 1 execution of the additional cycle from the state that visits location *L1* for the first time. This zone can be described by $y \in [0,2] \wedge z \in (6,\infty] \wedge z - y \in (6,\infty]$. Indeed, this symbolic state swallows all fragmented symbolic states that are reached after more than 1 executions of the acceleratable cycle.

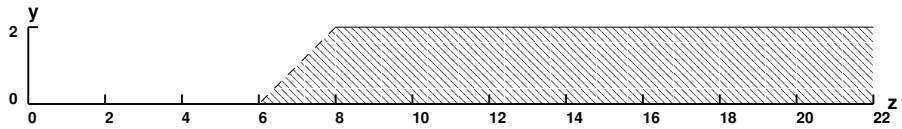

Figure 2.4: The zone that results from execution of the additional cycle from the state that visits location *L1* for the first time.

When we check whether location *finished* is reachable in model P and P_A as a function of the value of the constant L (using a breadth-first search order to enforce that the additional cycle is executed very early in the exploration), then we see that the time and space needed by both UPPAAL and KRONOS [107] for model P rises at least linearly while they are constant for model P_A. Theorem 2.14 explains this phenomenon. We claim that in practice the accelerated model performs much better than the normal model in both space and time.

2.5 Experimental Results

During previous work a compiler has been build which translates UPPAAL models to (i) executable LEGO Mindstorms code and (ii) another UPPAAL model of the run-time behavior of the executable code [54]. We constructed a very small example to illustrate how our acceleration technique can be used to speed-up the verification of the run-time behavior.

[4] Zones of a single location that form a convex union are not united in the current release of UPPAAL (3.4.11 and 3.5.9) since this is computationally expensive.

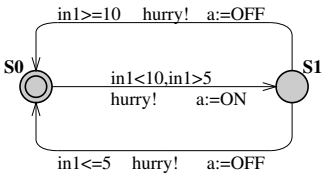

Figure 2.5: Process P_0.

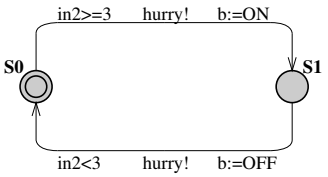

Figure 2.6: Process P_1.

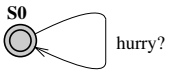

Figure 2.7: The hurry dummy.

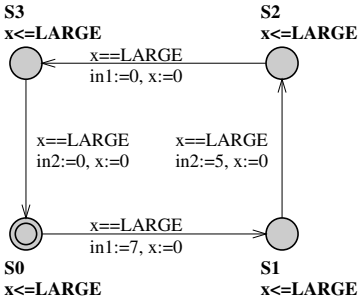

Figure 2.8: The environment.

Consider the processes P_0 and P_1 in Figures 2.5 and 2.6. These processes model a reactive program, called B, which controls two actuators and uses two sensors. Process P_0 uses sensor 1 (whose value is modeled by the variable *in1*) and actuator A (whose mode is modeled by the variable a). Similarly, process P_1 uses sensor 2 and actuator B. Initially, both actuators are off. If the sensor value of sensor 1 becomes between 5 and 10, process P_0 switches actuator A on. If the sensor value leaves this region, then process P_0 switches actuator A off again. Process P_1 functions in a similar manner.

Figures 2.7 and 2.8 depict the environmental processes. The hurry dummy provides an always enabled synchronization over the urgent channel *hurry*. The effect of this is that edges labeled with *hurry!* are taken as soon as they are enabled. Note that all edges of P_0 and P_1 use this channel, which models instantaneous reactions of these automata on the environment. The environment periodically updates the sensor values with a pace as expressed by the constant *LARGE*.

After compilation of the model, we obtain the symbolic byte code program shown in Figure 2.9. There are three kinds of instructions present. First, there are assignments, e.g., *v[0]:=0* and *actmode[A]:=off*. The first assignment manipulates the internal variable with index zero. The second assignment manipulates the mode of actuator A. Second, there are "test and branch far" instructions, e.g., *tbf 0!=v[0], 51*. If the boolean expression *0!=v[0]* evaluates to *true*, then control is transferred to the instruction with address 51. Otherwise, control is transferred to the next instruction. Finally, there are "branch always far" instructions, e.g., *baf 14*. This instruction transfers control to the instruction with address 14.

```
0000  v[0] := 0
0005  v[1] := 0
0010  actmode[A] := off
0012  actmode[B] := off
0014  tbf 0!=v[0], 51
0022  tbf 10<=snsval[0], 48
0030  tbf 5>=snsval[0], 48
0038  v[0] := 1
0043  actmode[A] := on
0045  baf 106
0048  baf 106
0051  tbf 10==snsval[0], 67
0059  tbf 10>=snsval[0], 77
0067  v[0] := 0
0072  actmode[A] := off
0074  baf 106
0077  tbf 5==snsval[0], 93
0085  tbf 5<=snsval[0], 103
0093  v[0] := 0
0098  actmode[A] := off
0100  baf 106
0103  baf 106
0106  tbf 0!=v[1], 143
0114  tbf 3==snsval[1], 130
0122  tbf 3>=snsval[1], 140
0130  v[1] := 1
0135  actmode[B] := on
0137  baf 164
0140  baf 164
0143  tbf 3<=snsval[1], 161
0151  v[1] := 0
0156  actmode[B] := off
0158  baf 164
0161  baf 164
0164  baf 14
```

Figure 2.9: The executable byte code.

The byte code simulates one interleaving of P_0 and P_1. The processes execute action transitions in an alternating way in an infinite loop. This loop starts with the instruction at address 14, and ends with the *baf* instruction at address 164. The *tbf* instructions inside the loop implement the alternation between P_0 and P_1 and the guards on the edges.

The second product of compilation is a model of the run-time behavior of the byte code program shown in Figure 2.9. This model naturally contains the environmental processes of Figures 2.7 and 2.8. The processes P_0 and P_1, however, are replaced by an exact model of the run-time behavior of the generated byte code, which is depicted in Figure 2.10.

The "byte code process" of Figure 2.10 is constructed by concatenation of models of the individual instructions of the executable byte code program [70]. Location Si models the $i + 1$-th instruction. For example, location *S0* models the first instruction, which is the assignment *v[0]:=0*. Clock x is used to model the duration of the instruction, which in this case is between 40 and 50 time units. The

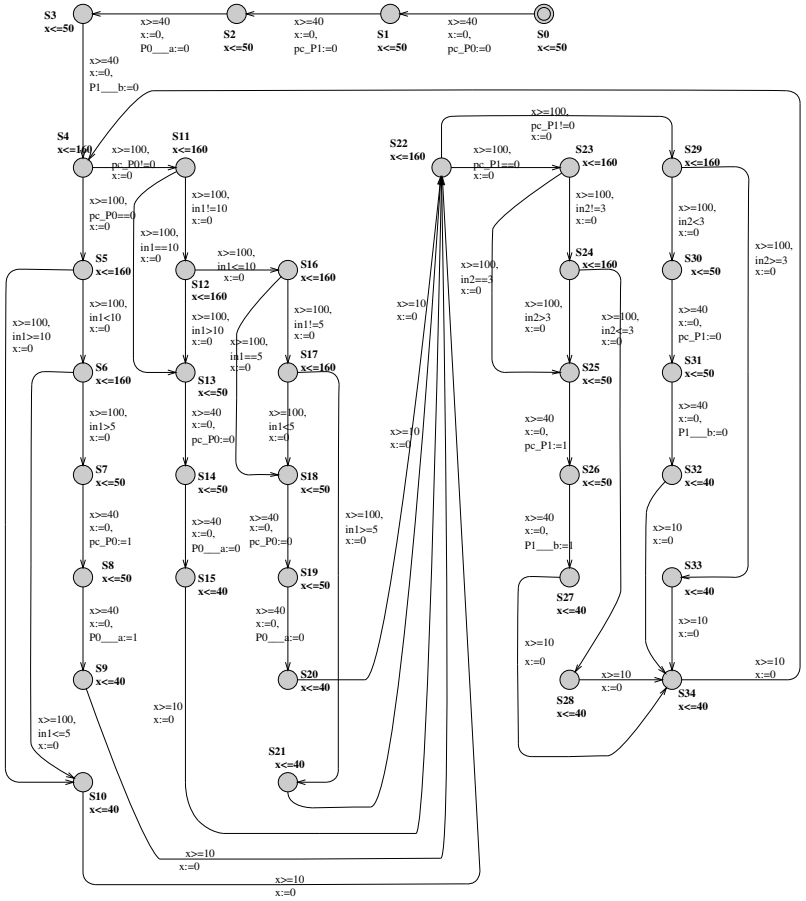

Figure 2.10: The UPPAAL model of the byte code program.

actual assignment is performed on exit of the location.

Note that every cycle in the byte code process is an acceleratable cycle. Our theory of exact acceleration, however, has been developed for single timed automata, and not for networks of timed automata that have been extended with bounded integer variables and concepts like urgency and commitment. We claim, however, that exact acceleration of single components in such an extended model can be achieved in the following way: (i) action transitions of a component are disabled if some other component is in its additional cycle, and (ii) an additional cycle can only be started if the clock that is used for it is equal to 0. A detailed proof of correctness of the acceleration of single components in a parallel composition is not presented here. Note, however, that the sketched approach certainly gives an over-approximation of the reachable state space.

We implemented our theory of exact acceleration in the compiler to accelerate *idle cycles* of the byte code processes. These idle cycles occur when no transi-

tions of the source processes of the byte code are enabled. In this situation, the byte code process tests all guards, but finds none satisfied. As a result, the byte code process exhibits useless busy-waiting behavior. To demonstrate the effect of this automatic application of exact acceleration, we checked two properties for the generated model of the run-time behavior of the byte code, namely $\mathbf{AG}(\mathit{true})$ and $\mathbf{EF}(\mathit{Env.S3})$. The first property is used to explore all reachable states and the second property is used to explore only a part of the reachable state space. The time and memory consumption for these two properties have been measured as a function of the value of the constant *LARGE* for the unaccelerated model B and for the model B_A in which four idle cycles are accelerated. Tables 2.1 and 2.2 contain the results, which show that the resource consumption is not insensitive to the value of the constant *LARGE* as in the theoretical example. For both complete and partial exploration of the state space, however, there still is a significant improvement.

Table 2.1: Run-time data of the regular model B and the accelerated model B_A for the property $\mathbf{AG}(\mathit{true})$ using a breadth-first search order.

	B		B_A	
LARGE	mem [kB]	t [s]	mem [kB]	t [s]
10^3	1120	0.03	1080	0.03
10^4	2396	0.07	1084	0.04
10^5	5924	2.80	4068	1.33
10^6	36204	752	18928	376

Table 2.2: Run-time data of the regular model B and the accelerated model B_A for the property $\mathbf{EF}(\mathit{Env.S3})$. Both the breadth-first and the depth-first search order have been measured for the regular model.

	B (bf)		B (df)		B_A (bf)	
LARGE	mem [kB]	t [s]	mem [kB]	t [s]	mem [kB]	t [s]
10^3	1144	0.02	720	0.02	800	0.03
10^4	2140	0.03	968	0.02	1084	0.03
10^5	4300	1.29	1140	0.04	1908	0.04
10^6	20244	311	5416	3.01	3268	1.22
10^7	-	-	32772	712	10900	279

2.6 Conclusions

We have presented an exact acceleration technique for forward symbolic reachability analysis of timed automata. Our technique is applicable to a subset of timed automata, namely those that contain acceleratable cycles. We append an extra cycle to the timed automaton that in a single execution computes the result of the

iterated execution of the acceleratable cycle in the original automaton. Whether or not a cycle is acceleratable, and the form of the additional cycle are easily computable from the syntax of the timed automaton. This makes the technique readily applicable in existing model checkers.

We have proved that our syntactic adjustment is exact with respect to reachability properties and that it will speed-up forward symbolic reachability analysis with a breadth-first search order. An example in which the technique is automatically applied demonstrates that it can be quite effective.

As future work it would be interesting to investigate the weakening of the constraints on acceleratable cycles as used in this chapter. Furthermore, we would like to evaluate the practical usefulness of this technique by applying it to a number of examples. Therefore, it would be convenient to have a tool that detects acceleratable cycles and accelerates them.

Acknowledgements. The authors thank Oliver Möller for his suggestions concerning the fragmentation problem, and Jozef Hooman for valuable comments on earlier versions of this chapter.

Chapter 3

Enhancing Uppaal by Exploiting Symmetry

MARTIJN HENDRIKS

Abstract. Efficiency is one of the major concerns in the world of model checking. Consequently, many techniques to optimize the time and space usage of model checking algorithms have been invented. One of these techniques is reduction of the searchable state space through arguments of *symmetry*. This technique can be very profitable and has been implemented in various model checkers. This paper proposes an enhancement of UPPAAL with symmetry reduction. We adopt the theory of symmetry of Ip and Dill and their scalarset data type, as implemented in the model checker MURφ. The main result of this chapter is a soundness proof of our symmetry enhancement, which does not follow trivially from the work of Ip and Dill since the description languages of UPPAAL and MURφ, which are used to detect the symmetries, are quite different.

3.1 Introduction

Model checkers emerge as practical tools for the mechanical verification of all kinds of systems [37]. In this approach, a model of the system to be verified and the verification properties serve as input to the tool, which consequently computes whether or not the model satisfies the specification. Nowadays, many model checkers are available, ranging from model checkers for JAVA source code [39] to model checkers for timed automata [16, 107].

Despite the relative ease of use of model checkers, they are not applied on a large scale. An important reason for this is that they must cope with the *state space explosion* problem. This is the problem of the exponential growth of the state space as models become larger. This growth often renders the mechanical verification of realistic systems practically impossible: there just is not enough time or memory available.

One possible approach to the huge resource requirements of model checkers consists of finding more efficient *techniques* to explore the state space. The exploitation of behavioral symmetries is such a well-known technique that has been successfully implemented for various model checkers, e.g., MURφ [45, 69], SMV [83] and SPIN [65, 26]. Especially the exploitation of *full* symmetries in a model can be profitable, since its gain can approach a factorial magnitude.

There are two main problems that need to be solved before symmetry reduction

This chapter is an almost literal copy of [55].

can be implemented. First, one has to statically detect symmetries in the system description. Second, during the state space exploration, one must decide whether or not some discovered state has already been encountered in the past. This decision should of course take symmetry into account. It has been shown that this so-called *orbit problem* is – in general – at least as difficult as testing for graph-isomorphism [38] and, unfortunately, there are no known polynomial algorithms for this last problem. This does not necessarily mean that symmetry reduction is a lost case, since we can always try to revert to sub optimal, but still feasible, solutions. Moreover, it might happen that the considered instance of the orbit problem is not so difficult.

This paper proposes a symmetry enhancement of UPPAAL, a model checker for networks of timed automata, by applying the theory of symmetry of Ip and Dill and adding their scalarset data type [69]. The main result of this paper is a soundness proof of our symmetry enhancement. More precisely, we prove that certain permutations on the states are sound with respect to reachability properties: if we have seen a state s, then we can conclude that we have seen all states which are obtainable by applying these permutations to s. Thus, we can use the permutations to reduce the amount of states which need to be explored by the model checking algorithm.

Our main result does not follow trivially from the work of Ip and Dill since the description languages of UPPAAL and MURφ, which are used to detect the symmetries, are quite different. This difference disallows us to apply Ip and Dill's soundness proof in a straightforward manner without loosing confidence in its validity. Since formal methods require mathematical precision, we are forced to construct a new proof for the soundness of symmetry reduction in UPPAAL.

This work is directly motivated by attempts to verify various distributed systems, which clearly exhibit full symmetry, using UPPAAL. For example, Fischer's mutual exclusion protocol (see, for instance, [1]) for 12 or more processes is practically unverifiable. Similarly, a simple model of a CSMA/CD protocol (see, for instance, [107]) is practically unverifiable when 13 or more processes are considered. The state space explosion problem is felt more directly during the attempts of verifying a distributed agreement algorithm [10]. It is very difficult to verify the smallest interesting instance of the algorithm (three processes). Hopefully, implementation of symmetry enhanced UPPAAL can help us to overcome these boundaries.

Outline. In Section 3.2 we summarize the theory of Ip and Dill for symmetry reduction. In Section 3.3 we extend the description language of UPPAAL with the scalarset data type, and with multidimensional arrays of integer variables and channels. We illustrate the extended syntax by modeling Fischer's mutual exclusion protocol. Moreover, we give a formal definition of these SUPPAAL models, and we explain their semantics. In Section 3.4 we extract the automorphisms from the system description, and we give the soundness proof. Finally, in Section 3.5 we summarize this paper and we discuss future work.

3.2 A Theory of Symmetry

In this section we summarize the theory of symmetry developed by Ip and Dill [69]. They consider *state graphs*, which are tuples containing a set Q of states, a set $Q_0 \subseteq Q$ of initial states, a transition relation $\Delta \subseteq Q \times Q$ and a unique error state, which we omit in our presentation[1]. A state $q \in Q$ is *reachable*, iff a sequence $q_0, q_1, ..., q_{n-1}$ exists, such that $q_0 \in Q_0$, $q = q_{n-1}$, $q_i \in Q$ for all $0 \leq i < n$, and $(q_i, q_{i+1}) \in \Delta$ for all $0 \leq i < n-1$.

We assume the existence of a set of *state properties* Φ, for whose elements we can decide whether they are true or false in some state. If a state property $\phi \in \Phi$ is true in state q, then we denote this by $q \models \phi$. Also, we assume that for each state q the set $succ(q) = \{q' | (q, q') \in \Delta\}$ is finite and effectively computable. Figure 3.1 depicts a standard forward exploration algorithm, which (semi) decides whether or not a state is reachable which satisfies some given state property ϕ.

```
passed := ∅
waiting := Q0
while waiting ≠ ∅ do
        get q from waiting
        if q ⊨ φ then return YES
        else if q ∉ passed then
                add q to passed
                waiting := waiting ∪ succ(q)
        fi
od
return NO
```

Figure 3.1: Standard forward reachability analysis.

The algorithm starts by adding the initial states to the *waiting* set. Then it enters a loop that processes all the states in the *waiting* set in the following way. If some waiting state q satisfies the state property ϕ, then the algorithm returns YES. Otherwise, it checks whether q has already been seen. If this is the case, $q \in$ *passed*, then the algorithm discards q and gets a new state from the *waiting* set. If q has not yet been encountered, then it is added to the *passed* set and all its successors are added to the *waiting* set. If the state space – the set Q – is finite, then this algorithm halts. Otherwise, it may not halt.

Ip and Dill define symmetry within a state graph as a graph automorphism different from the identity relation.

Definition 3.1 (Automorphism) *Let (Q, Q_0, Δ) be a state graph. A* graph automorphism *on this state graph is a bijection $h : Q \to Q$ such that*

(i) $q \in Q_0$ iff $h(q) \in Q_0$ for all $q \in Q$, and

[1] Omitting the error state does not change the validity of Ip and Dill's results [69].

(ii) $(q_1, q_2) \in \Delta$ *iff* $(h(q_1), h(q_2)) \in \Delta$ *for all* $q_1, q_2 \in Q$.

For any set of graph automorphisms H, the closure of $H \cup \{\mathbf{id}\}$, where **id** is the identity function, under inverse and composition, denoted by $\mathcal{C}(H)$, is a group. Such a *symmetry group* $\mathcal{C}(H)$ induces a relation $\approx_H \subseteq Q \times Q$ such that $q_1 \approx_H q_2$ iff there exists an $h \in \mathcal{C}(H)$ such that $h(q_1) = q_2$. This relation is an equivalence relation and we let $[q]$ denote the equivalence class of state q. Using these equivalence classes we can define a quotient graph.

Definition 3.2 (Quotient graph) *Let* $A = (Q, Q_0, \Delta)$ *be a state graph and let* $\mathcal{C}(H)$ *be a symmetry group for* A. *The quotient graph induced by* $\mathcal{C}(H)$ *is the graph* $A_{\approx_H} = (Q', Q'_0, \Delta')$, *where* $Q' = \{[q] \mid q \in Q\}$, $Q'_0 = \{[q] \mid q \in Q_0\}$ *and* $\Delta' = \{([p], [q]) \mid (p, q) \in \Delta\}$.

Ip and Dill observed that a quotient graph can be used to check reachability properties: q is reachable in A iff $[q]$ is reachable in $A_{\approx_H}$. Since the quotient graph is at most as large as the original state graph, and in many cases smaller, the use of the quotient graph can speed up the model checking process.

As already mentioned in the introduction, the two major problems that should be solved in the actual implementation of symmetry reduction are the following:

- We must detect a set of automorphisms from the system description. The corresponding symmetry group induces the (smaller) quotient graph.
- During the exploration of the state space, we must be able to decide whether or not two states are symmetric. Thus, for states q and q', we must decide whether or not $[q] = [q']$.

In order to protect the gain of using the quotient graph, the approaches to both problems should be computationally cheap. In the next sections, we add the well-known scalarset data type to UPPAAL in order to statically detect symmetries from the system description. As for the second problem, our strategy is to convert all explored states to a so-called *normal form* using the detected automorphisms. This normal form represents the equivalence class of the state. The only correctness requirement for our normal form operator θ is the following:

$$\forall_{q,q' \in Q} \ \big(\theta(q) = \theta(q') \ \Rightarrow \ [q] = [q']\big) \tag{3.1}$$

This criterion says that if two states have the same normal form, then they are contained in the same equivalence class. Note that if $\theta \in \mathcal{C}(H)$, then property (3.1) is certainly satisfied. If the implication of property (3.1) also holds the other way around, then the normal form operator is *canonical*.

The function $\Theta : 2^Q \rightarrow 2^Q$ converts a set of states to their normal forms in the regular way: $\Theta(Q) = \{\, \theta(q) \mid q \in Q \,\}$. We now can state a new forward exploration algorithm, depicted in Figure 3.2, which uses the normal form operator to take symmetry into account.

```
passed := ∅
waiting := Θ(Q_0)
while waiting ≠ ∅ do
        get q from waiting
        if q ⊨ φ then return YES
        else if q ∉ passed then
                add q to passed
                waiting := waiting ∪ Θ(succ(q))
        fi
od
return NO
```

Figure 3.2: Adding symmetry to the forward reachability analysis.

This new algorithm uses θ and Θ to convert all discovered states to their normal forms. If θ is canonical, then exactly the quotient graph will be explored. However, if θ is not canonical, then *at most* the original state graph will be explored.

Note that the symmetry reduction technique as shown in algorithm 3.2 only is sound if the state property ϕ is symmetric:

$$\forall_{q,q' \in Q} \ \big(q \approx_H q' \Rightarrow (q \models \phi \Leftrightarrow q' \models \phi)\big) \tag{3.2}$$

If this is not the case, then the normal form that is stored to represent a whole symmetry class might not satisfy ϕ while an element of the symmetry class does satisfy ϕ, which is not detected by the algorithm.

3.3 From Uppaal to SUppaal

The tool UPPAAL has been based on the theory of timed automata of Alur and Dill [7, 5]. In short, a UPPAAL model consists of a network of timed automata enhanced with (arrays of) bounded integer variables, which communicate through shared variables and by binary blocking synchronizations (see, for instance, the *help* menu in the tool itself and [16]). In this section we explain how we add the scalarset data type to the system description language of UPPAAL. We assume some knowledge about this description language, and in particular about the "templates" and their instantiation mechanism. Note that UPPAAL 3.2 has been taken as a basis for this work.

In section 3.3.1 we formally define the symmetry extension of the syntax of the system description language. In section 3.3.2 we give a formal definition of UPPAAL models enhanced with symmetry, which we call SUPPAAL models from now on. Moreover, we present a version of a model of Fischer's mutual exclusion protocol that has been adjusted for symmetry. Finally, in section 3.3.3 we explain the semantics of these models using the mathematical representation.

3.3.1 Adding the scalarset data type to Uppaal

In this section, we explain how we extend the syntax of the system description language of UPPAAL with scalarsets. We do not give the complete original syntax of this language here since it can be easily found in the help menu of the tool itself. At the end of this section we use the extended syntax to model Fischer's mutual exclusion protocol (see, e.g., [1]).

The additions we propose are split into three parts. First, we explain the additions to the declarations section, second, we explain the changes to template parameters, and third, we explain the additions to the process assignments section.

Additions to the declarations section

First, we add the scalarset data type to UPPAAL. This is a sub range of integers with fixed size, and whose elements can be arbitrarily permuted without changing the behavior of the system. First, we want to be able to declare scalarset types with different sizes in the global declaration sections of UPPAAL models as follows:

```
scalarset pid[3];
scalarset bid[5];
```

A scalarset α is the sub range $\{0, ..., |\alpha| - 1\}$, where $|\alpha|$ denotes the size of the scalarset. From now on, Ω is used as a set of scalarset names (each element of Ω is associated with a subrange of the natural numbers). We should be able to use scalarset names in declarations. For example:

```
int[0,1] input[bid];
```

The *scalarset array* `input` contains boolean values. Its size is fixed by the size of the scalarset, which in this case equals 5. In order to make optimal use of scalarsets we also add *multi dimensional* arrays of bounded integer variables and channels. For example:

```
int dim3[pid][bid][7];
chan cd[5][pid];
```

The three-dimensional integer array `dim3` thus contains $3 \times 5 \times 7$ elements, and `cd` is a two dimensional array of channels with 15 elements.

Next, we formally state the changes to the original syntax of UPPAAL 3.2 to facilitate the mechanisms sketched above (see the help menu of UPPAAL for the complete original syntax definition). First, we add the scalarset type to the syntax:

```
Declarations ::= ( NewDecl ';' )*

NewDecl ::= Decl  |  'scalarset' ID '[' CExpr ']'
```

The second formal adjustment to the original syntax is the redefinition of the grammar for `IL`:

```
IL ::= ILID ( ',' ILID )*

ILID ::= ID ( '[' CExpr ']' )*
```

This renewed definition of IL allows us to declare (multi-dimensional) arrays of channels. Note that with this syntax we can also declare (multi-dimensional) arrays of clocks. However, for reasons of simplicity we do not allow this. Finally, we redefine the variable identifiers:

```
VID ::= ID ( '[' (CExpr | ID) ']' )*  |  ID ':=' CExpr
```

The renewed definition of VID allows us to declare multi-dimensional arrays of integers. The second appearance of the non-terminal ID in the first rule of the definition of VID must be bound to a scalarset. Thus, we can use a scalarset type to index arrays.

Additions to the template parameters

Apart from using scalarset types to declare variables and to index arrays, we also want to use them for the instantiation of templates. Consider for example the well-known Fischer mutual exclusion protocol. It contains n processes, which only differ in a unique process identifier, which can be modeled by the scalarset pid. The UPPAAL model only contains a single template, say P, of a generic Fischer process which takes a process identifier as argument. All processes are created by instantiation of this template. We propose the following syntax for the template parameters of P:

```
process P ( scalarset pid; )
```

For the sake of simplicity we only allow templates to be instantiated with scalarsets. Thus, we redefine the syntax for template parameters as follows:

```
Param ::= ( 'scalarset' ID (',' ID)* )*
```

Note that this definition allows an empty parameter list.

Additions to the process assignments section

We explained above that we can model the unique process identifiers of the Fischer protocol as a scalarset, say pid. Additionally, we can instantiate the above mentioned template P with the elements of this scalarset. We propose the following syntax for this instantiation in the process assignments section:

```
FischerProcs := P(pid);
```

Informally, this means that the "object" FischerProcs is the parallel composition of |pid| instantiations of template P with *all* elements of the scalarset pid. Moreover, we also propose *layered* instantiation using the scalarsets. Assume that pid and bid are scalarsets, then

```
Procs1 := P(pid);
Procs2 := B(pid,bid);
```

creates the process Procs1 as the parallel composition of |pid| instances of template P and Procs2 as the parallel composition of |pid| × |bid| instances of template B. In general, we redefine the syntax for the process assignments, given by PAList, as follows:

```
PAList ::= (ID ':=' ID '(' [Values] ')' ';')*

Values ::= ID (',' ID)*
```

Of course, there are some syntactical and semantic restrictions for using the new process assignment construction. We do not elaborate on them here, since they are not very interesting and mostly straightforward.

Example: Fischer's mutual exclusion protocol

We can use the new syntax to model Fischer's mutual exclusion protocol (based on the model that is distributed with UPPAAL). We start with the global declarations of the SUPPAAL model:

```
scalarset process_id[3];
int id:=-1;
```

Next, the template `P` with the header `process P (scalarset pid)` and a local clock `x` is defined. This template has been depicted in Figure 3.3.

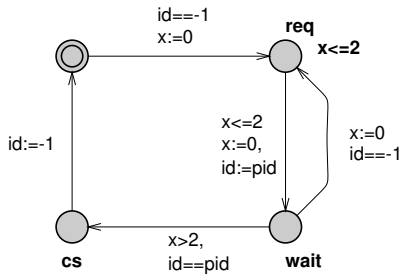

Figure 3.3: The Fischer process template.

The global idea of the protocol is that each "Fischer process" has a unique process identifier (elements of the scalarset `pid` in our model). As soon as a Fischer process wants to enter its critical section, it writes its own identifier in a global variable (`id` in our model). If the global variable still contains its identifier after a certain amount of time (2 time units in our model), then the Fischer process may enter its critical section. When it leaves its critical section, it resets the global variable to a neutral value (-1 in our model). The process assignments section consists of only the following line:

```
Procs := P(process_id);
```

As we explained in the previous section, the object `Procs` denotes the parallel composition of 3 Fischer processes (since the scalarset `process_id` has size 3). Note that the process identifiers range from 0 to 2, which ensures that -1 indeed is a neutral value for the global variable `id`. Finally, the system definition section only contains the line "`system Procs;`", which speaks for itself.

3.3.2 Mathematical Description of SUppaal Models

In the previous section we added multidimensional arrays and the scalarset data type to UPPAAL's system description language to obtain the system description language of SUPPAAL. In this section we give a mathematical representation of these SUPPAAL models, which can easily be derived from the new system description language.

To formally define a SUPPAAL model, we first need to define *guards*, *invariants* and *assignments* over the clocks and *guards* and *assignments* over the bounded integer variables.

Definition 3.3 (Clock guard) *A clock guard ϕ over a set of clocks X is defined by the grammar $\phi ::= x \sim n \mid x \sim y + n \mid \phi \wedge \phi$, where $x, y \in X$, $n \in \mathbb{N}$ and $\sim \in \{\leq, <, =, >, \geq\}$[2]. We let $CG(X)$ denote the set of all clock guards over X.*

Next, we define the set of invariants, which is a subset of the set of clock guards.

Definition 3.4 (Invariant) *An invariant ϕ over a set of clocks X is defined by the grammar $\phi ::= x \sim n \mid \phi \wedge \phi$, where $x \in X$, $n \in \mathbb{N}$ and $\sim \in \{\leq, <\}$. We let $Inv(X)$ denote the set of all invariants over X.*

Clock assignments reset clocks to an integer value greater than or equal to zero.

Definition 3.5 (Clock assignment) *A clock assignment ca over a set of clocks X is defined by the grammar $ca ::= x := n$, where $x \in X$ and $n \in \mathbb{N}$. We let $CA(X)$ denote the set of all clock assignments over X.*

Next, we define the integer expressions, which can be used as part of integer assignments, integer guards and synchronizations. For sake of simplicity, we do not allow arrays to index arrays.

Definition 3.6 (Integer expression) *An integer expression, IExpr, over a set of variables V and a set of scalarset names Ω is defined by the grammar*

$$\textit{SIExpr} \quad ::= \quad z \quad | \quad \alpha \quad | \quad v \quad | \quad (\textit{SIExpr} \oslash \textit{SIExpr})$$

$$\textit{IExpr} \quad ::= \quad SIexpr \quad | \quad v[\textit{SIExpr}]^{+}$$

where $z \in \mathbb{Z}$, $\alpha \in \Omega$, $v \in V$, and $\oslash \in \{/, *, +, -\}$. We let $IX(V, \Omega)$ denote the set of all integer expressions over V and Ω. Elements of SIExpr are called simple integer expressions.

An integer guard consists of a conjunction of comparisons between two integer expressions.

[2]The constant *true* can be defined by $x \geq 0$.

Definition 3.7 (Integer guard) *An integer guard ϕ over a set of variables V and a set of scalarset names Ω is defined by the grammar $\phi ::= IExpr \sim IExpr \mid \phi \wedge \phi$, where $\sim \in \{\leq, <, =, \neq, >, \geq\}$. We let $IG(V, \Omega)$ denote the set of integer guards over V and Ω.*

An integer assignment assigns the value of some integer expression to an integer variable, or to an entry of a (multi dimensional) integer array.

Definition 3.8 (Integer assignment) *An integer assignment ϕ over a set of variables V and a set of scalarset names Ω is defined by the following grammar $\phi ::= v[SIExpr]^* := IExpr$ where $v \in V$. We let $IA(V, \Omega)$ denote the set of all integer assignments over V and Ω.*

Finally, we define the set of synchronizations over a set of synchronization labels Σ, a set of variables V, and a set of scalarset names Ω.

Definition 3.9 (Synchronization) *A synchronization ϕ over a set of labels Σ, a set of variables V and a set of scalarset names Ω is defined by the grammar*

$$\phi ::= \quad \tau \quad | \quad \sigma[SIExpr]^*! \quad | \quad \sigma[SIExpr]^*?$$

where τ denotes the "empty" synchronization and $\sigma \in \Sigma$. We let $Sync(\Sigma, V, \Omega)$ denote the set of all synchronizations over Σ, V, and Ω.

For instance, if $s \in \Sigma$ and $\alpha \in \Omega$, then $s[\alpha]! \in Sync(\Sigma, V, \Omega)$. Such a synchronization does not mean that α is send over the channel s. Instead, it expresses the blocking synchronization over the $\rho(\alpha)$*-th* element of the array of channels s (ρ assigns a value to scalarset names in the context of processes; this becomes clear below). This construct is motivated by models in which the symmetric components are modeled by the parallel composition of synchronizing processes, e.g., a computation process and a broadcast process.

A SUPPAAL model consists of a set of global integer variables, V^g, a set of global clocks, X^g, a set of channels, Σ, a set of scalarset names, Ω, and multiple SUPPAAL processes with local variables and clocks. The processes are created by instantiation of templates, and therefore we proceed with the definition of these templates over V^g, X^g, Ω and Σ.

Definition 3.10 (SUPPAAL template) *A SUPPAAL template over V^g, X^g, Ω and Σ is a tuple $T = (L, L^0, lt, X, V, S, I, vt, init, E)$, where*

- *L is a finite set of locations,*
- *$L^0 \in L$ is the initial location,*
- *$lt : L \rightarrow \{regular, urgent, committed\}$ assigns a type to every location,*
- *X is a finite set of local clocks (assume $X^g \cap X = \emptyset$),*

- V *is a finite set of local bounded integer variables (assume* $V^g \cap V = \emptyset$*),*
- $S \subseteq \Omega$ *is a finite set of scalarset names,*
- $I : L \to Inv(X^g \cup X)$ *assigns invariants to locations,*
- *vt* $: V \to (\mathbb{N} \cup S)^*$ *assigns a type to every local variable,*
- *vs* $: V \to (\mathbb{Z} \times \mathbb{Z})$ *assigns a finite domain to every local variable,*
- *init* $: V \to \mathbb{Z}^+$ *initializes every local variable, and*
- $E \subseteq L \times Sync(\Sigma, V \cup V^g, S) \times G \times A \times L$ *is a set of edges, where*
  - G *is a pair of guards in* $IG(V \cup V^g, S) \times CG(X \cup X^g)$*, and*
  - A *is a pair of assignments in* $(IA(V \cup V^g, S))^* \times 2^{CA(X \cup X^g)}$.

It is relatively straightforward to construct the mathematical SUPPAAL templates from the system description language. The only difficulty might occur when constructing the set S (and thereby *vt* and E). During the construction the template parameter is replaced by the scalarset it mimics. In the example of the Fischer protocol, this means that the set S for the template `P` contains only one scalarset, namely `process_id`, and the name `pid` in the edges of the template has been replaced by `process_id`.

The *vt* function can be explained by an example. Assume that v is a variable and that $vt(v) = (3, \alpha, 8)$, where α is a scalarset in S. This means that the variable v is a three dimensional array. Its first dimension is a regular dimension with size 3, its second dimension is indexed by scalarset α (its size is also determined by α, see below), and its third dimension also is a regular dimension, but with size 8. If $vt(v)$ equals the empty sequence, denoted by ϵ, then v is a regular variable.

The *init* function initializes the local variables. Our representation does not include the "decoding" scheme needed for arrays. However, this does not matter, since array entries are initialized to 0 by default.

We use indices to refer to the specific parts of templates. E.g., if T_i denotes a template, then X_i denotes the set of local clocks of T_i. With the previous definitions of SUPPAAL templates we are ready to define SUPPAAL models.

Definition 3.11 (SUPPAAL **model)** *A* SUPPAAL *model is defined by a tuple* $M = (\Omega, s, V^g, vt, init, X^g, \Sigma, ct, \mathbb{T})$*, where*

- Ω *is a finite set of scalarset names,*
- $s : \Omega \to \mathbb{N}$ *defines the size of each scalarset (*$s(\alpha)$ *is also written as* $|\alpha|$*),*
- V^g *is a finite set of global integer variables,*
- *vt* $: V^g \to (\mathbb{N} \cup \Omega)^*$ *assigns a type to every global variable,*

- *vs* : $V^g \to (\mathbb{Z} \times \mathbb{Z})$ *assigns a finite domain to every global variable,*
- *init* : $V^g \to \mathbb{Z}^+$ *initializes every global variable,*
- X^g *is a finite set of global clocks,*
- Σ *is a finite set of communication channels,*
- *ct* : $\Sigma \to \{$*regular*, *urgent*$\}$ *assigns a type to every channel, and*
- $\mathbb{T}$ *is a finite set of templates over* V^g, X^g, Ω *and* Σ.

The tuple M contains all information needed to construct the set of actual processes in the system, which defines the semantics of the model. We define these SUPPAAL processes as follows.

Definition 3.12 (SUPPAAL process) *Let M be a* SUPPAAL *model as above. A* SUPPAAL *process of M is a tuple $A = (T, \rho)$, where $T \in \mathbb{T}$ and $\rho : S \to \mathbb{N}$, such that $0 \leq \rho(\alpha) < s(\alpha)$ for all scalarset names $\alpha \in S$.*

As with templates, we use indices to refer to the components of processes. Thus, if A_i is a process, then T_i is its template, and V_i is the set of local variables of the process' template.

Note that SUPPAAL processes of M that originate from the same template have equal sets of local variables, clocks and scalarsets. To simplify the explanation of the semantics of M, which we define in the next section, we "flatten" the presentation. That is, we define the set of processes $\mathbb{A}$ associated with M as uniquely renamed SUPPAAL processes of M:

$$\mathbb{A} = \{\, \mathit{rename}_{(T_i,\rho_i)}((T_i, \rho_i)) \mid (T_i, \rho_i) \text{ is a SUPPAAL process of } M \,\} \tag{3.3}$$

By subscribing the *one-to-one* renaming functions with (T_i, ρ_i), we want to express that these renaming functions are unique in the sense that the local clocks, variables and constants of different processes share no elements. This allows us to merge the sets of clocks, variables and variable type functions of all processes of a SUPPAAL model. From now on, *Var* denotes the set of all variables, *tVar* denotes the set of all variable type mappings and *Clock* denotes the set of all clocks. As will become clear later, this assumption, which can be made without loss of generality, is very convenient for the definition of the semantics of the model.

We can easily extract the mathematical description of the model from any SUPPAAL system description. From this mathematical description, we can generate the set of processes $\mathbb{A}$ (thereby choosing suitable renaming functions). Moreover, we can derive the partial equivalence function $equiv_{ij}^{V} : V_i \to V_j$ for every pair of *processes* A_i and A_j:

$$\mathit{equiv}_{ij}^{V}(a) = \begin{cases} b & \text{if } T_i = T_j \text{ and } \mathit{rename}^{-1}_{(T_i,\rho_i)}(a) = \mathit{rename}^{-1}_{(T_j,\rho_j)}(b) \\ \uparrow & \mathit{otherwise} \end{cases}$$

This equivalence function links the local variables of processes which originate from the same template. Similarly, we can derive the partial equivalence function $equiv_{ij}^{X} : X_i \rightarrow X_j$ for local clocks. It is straightforward to see that if $equiv_{ij}(a)$ is defined, then $equiv_{ji} \circ equiv_{ij}(a) = a$.

Syntactical Restrictions on SUPPAAL Models

The mathematical description of a SUPPAAL model as defined previously should satisfy a number of restrictions to be syntactically correct. For instance, a reference to an array of integers must contain the right number of dimensions given by the *tVar* function. We do not elaborate on these well-understood restrictions here. Instead, we discuss restrictions concerning the newly introduced scalarsets.

First, we need to introduce the concept of *well-formed* integer expressions. Therefore, we need the help of a projection $[\,]_i$ which selects elements from arrays or sequences. This projection function is defined as follows:

$$[(e_0, e_1, ..., e_n)]_i = \begin{cases} e_i & \text{if } 0 \leq i \leq n \\ \uparrow & \text{otherwise} \end{cases}$$

Informally, an integer expression is well-formed if it is not an array, or if it is an array of which all scalarset dimensions are indexed by scalarset constants of the same type.

Definition 3.13 (Well-formedness) *Let $exp \in IX(V^g \cup V_i, S_i)$ be an integer expression of some process. We call exp well-formed, if exp is a simple integer expression, or if $exp = a[i_0]...[i_m]$ such that if $[\mathit{tVar}(a)]_k = \alpha$, then $i_k = \alpha$ for all $\alpha \in \Omega$ and for all $0 \leq k \leq m$.*

This concept of well-formedness can easily be lifted to well-formedness of synchronizations, integer assignments, integer guards and edges. For instance, assume that we have a SUPPAAL model with a scalarset `id` with size 3, and a variable array `a`, such that *tVar*(`a`) = (`id`). This means that `a` is a one-dimensional array indexed by the scalarset `id`. Moreover, there is a template in this model which is parameterized with this scalarset, and which has local integer variable `v`. Figure 3.4 depicts a well-formed edge of this template, and Figure 3.5 depicts a non well-formed edge.

We also distinguish a special subset of non well-formed integer expressions and synchronizations.

Definition 3.14 ((α, n)-malformedness) *Let $a[i_0] \ldots [n] \ldots [i_m]$ be an non well-formed integer expression or synchronization. If the $n \in \{0, ..., s(\alpha) - 1\}$ is the*

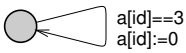

Figure 3.4: A well-formed edge.

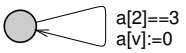

Figure 3.5: A non well-formed edge.

unique *cause of this since it indexes an α dimension, then we call this integer expression or synchronization (α, n)-malformed.*

Again, we can easily lift this concept to edges: an edge is (α, n)-malformed, iff it contains (α, n)-malformed integer expressions or synchronizations. For instance, assume the same context as associated with the figures above and consider Figure 3.6, which contains three $((\texttt{id}, n))$-malformed edges: one for every $n \in \{0, 1, 2\}$. Note that the edge of Figure 3.5 is *not* (α, n)-malformed for any α or n.

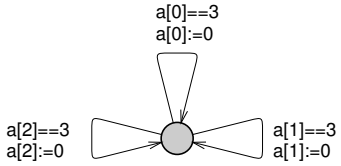

Figure 3.6: $(\texttt{id}, n)$-malformed edges (where $n \in \{0, 1, 2\}$).

Now we can state the restrictions concerning scalarsets in a SUPPAAL model. In short, there may be no symmetry breaking operations on scalarsets and we have formulated the following restrictions to achieve this:

(1) Consider a scalarset $\alpha \in S_i$ of process A_i. We can use α in only three ways:

   a) Assign the scalarset to a regular variable: $v := \alpha$ may appear in process A_i. (See the assignment `id:=pid` in Figure 3.3).

   b) Use the scalarset in guards: $v = \alpha$ and $v \neq \alpha$ may appear in process A_i. (See the guard `id==pid` in Figure 3.3).

   c) We can use α to index α dimensions of arrays: $a[i_0] \ldots [\alpha] \ldots [i_n]$, where α indexes dimension k and $a \in \mathit{Var} \cup \Sigma$, may appear in process A_i, if $[\mathit{tVar}(a)]_k = \alpha$. This means that well-formed integer expressions may appear in our model.

   Thus, the scalarsets in processes may not be used in arithmetical expressions, clock guards and invariants.

   We can compute a set of variables used_α for each scalarset α as follows: $a \in \mathit{used}_\alpha$ if and only if there exists an assignment $a := \alpha$ or there exists a guard $a = \alpha$ or $a \neq \alpha$ in the model.

(2) Let $v \in \mathit{used}_\alpha$ for some scalarset α. For instance, the global variable `id` in the Fischer protocol example at the end of section 3.3.1 is an element of $\mathit{used}_{\texttt{process\_id}}$.

a) v is initialized to a value $\notin \{0, ..., |\alpha| - 1\}$.

b) $\mathit{used}_\alpha \cap \mathit{used}_\beta = \emptyset$ for all scalarsets α and β.

c) v can *only* be used in assignments and guards as in (1a) and (1b) or we can assign an integer value z to v, or we can use v in a guard of the form $v = z$ or $v \neq z$, such that $z \notin \{0, 1, ..., |\alpha| - 1\}$.

Thus, v can neither be used in arithmetical expressions, nor can it be used to index arrays. Note that our Fischer example satisfies these conditions: `id` is initialized to -1, the Fischer template only assigns -1 or the scalarset constant to `id`, and it compares `id` only to -1 or to the scalarset constant.

(3) An edge $e = (src, \sigma, g, a, dst) \in E_i$ of a SUPPAAL model should either be well-formed, or it should be (α, n)-malformed, such that:

- the edge is not (β, m)-malformed for any $\beta \neq \alpha$ or $m \neq n$, and
- for every $n' \in \{0, ..., |\alpha| - 1\} \setminus \{n\}$ there exists an (α, n')-malformed edge $e' = (src, \sigma', g', a', dst) \in E_i$ such that σ', g' and a' only syntactically differ from σ, g and a in that the appearances of n as α dimension index have been replaced by n'.

For instance, the example belonging to Figure 3.6 satisfies these constraints, since every $(\texttt{id}, n)$-malformed edge is not $(\texttt{id}, m)$-malformed for an $m \neq n$ (for instance, an edge labeled with the guard `a[1]==3` and the assignment `a[2]:=0` is not allowed). Moreover, there are the three required "equivalent" $(\texttt{id}, n)$-malformed edges: one for each $n \in \{0, 1, 2\}$.

Note that this restriction allows processes to "reset" scalarset dimensions of arrays without breaking the symmetry, which can be very convenient if not necessary for the modeling of many systems.

The restrictions above are very similar to those imposed by Ip and Dill, except for item (2). In contrast with their theory, we allow the assignment of a "scalarset type" to a regular variable (these variables are caught by the *used*-sets). Our motivation is based on the Fischer protocol, which would not fit in our framework without this extension, although it clearly exhibits full symmetry. With the previous restrictions we are able to prove in section 3.4 that our proposed symmetry reduction technique is sound.

3.3.3 Semantics of SUppaal Models

The semantics of a SUPPAAL model is, as with regular timed automata, defined by an infinite transition system which originates from the set of (renamed!) SUPPAAL processes $\mathbb{A}$. The states of this transition system are defined as follows:

Definition 3.15 (State) *A state of a* UPPAAL *model containing n processes is a tuple $(\vec{l}, v, \nu)$, where*

- *$\vec{l}$ is the location vector, such that $l_i \in L_i$ for all $0 \leq i \leq n-1$,*
- *$v : Var \rightarrow \mathbb{Z}^*$ is the variable valuation, which maps every variable to a value (or a tuple of values in case of an array)[3], and*
- *$\nu : Clock \rightarrow \mathbb{R}_+$ is the clock valuation which maps every clock to a non-negative real number including zero.*

The set of variable valuations for some model is denoted by Λ, the set of clock valuations for some model is denoted by Γ, and the set of all states is denoted by $\mathbb{S}$.

This definition explains the need for the renaming functions in definition 3.11. If we allow equal local names of variables or clocks in processes, then we would need a clock and variable valuation for every process.

There are three kinds of transitions between states of a UPPAAL model. Before we define these, we specify how the assignments and guards are interpreted over the clock and variable valuations. Since the integer guards and integer assignments might contain scalarsets, we need the context of a SUPPAAL process (more precisely, the scalarset valuation ρ) for evaluation.

$$eval : IX(Var, \Omega) \times \mathbb{A} \times \Lambda \hookrightarrow \mathbb{Z}$$

$$eval : CG(Clock) \times \Gamma \rightarrow \{true, false\}$$

$$eval : IG(Var, \Omega) \times \mathbb{A} \times \Lambda \hookrightarrow \{true, false\}$$

These functions are defined in the usual way. Using these evaluation functions, we can easily define the integer assignment execution function and the clock reset execution function, whose types are given below.

$$exec : (IA(Var, \Omega))^* \times \mathbb{A} \times \Lambda \hookrightarrow \Lambda$$

$$exec : 2^{CA(Clock)} \times \Gamma \rightarrow \Gamma$$

Now we are ready to define the transitions of a SUPPAAL model (we assume that there are n processes). The first kind of transition is a simple action transition in which an individual process executes an edge labeled with the "empty" channel τ:

Definition 3.16 (Simple action transition) *A tuple of states $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ is a simple action transition if an edge $(src, \tau, (\gamma_c, \gamma_v), (ac, av), dst) \in E_k$ exists, such that*

[3] We do not explicitly explain the encoding and decoding of these arrays, since it is enough to assume that this happens in a consistent way.

- $l_k = src$ *and* $l'_k = dst$ *and* $l'_j = l_j$ *for all* $j \neq k$,
- $eval((\gamma_c, \nu)) = eval((\gamma_v, A_k, v)) =$ *true,*
- $v' =$ *exec*$((av, A_k, v))$ *and* $\nu' =$ *exec*$((ac, \nu))$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l'_i), \nu')) =$ *true for all* $0 \leq i \leq n-1$, *and*
- *if there exists a* l_i *such that* $lt_i(l_i) =$ *committed, then* $lt_k(l_k) =$ *committed.*

The second kind of transition is a synchronizing action transition in which two processes simultaneously execute an edge with matching synchronization labels.

Definition 3.17 (σ action transition) *A tuple* $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ *of states is a* σ *action transition if there exists an edge* $(src, (\sigma, !), (\gamma_c, \gamma_v), (ac, av), dst) \in E_k$ *and an edge* $(src', (\sigma, ?), (\gamma'_c, \gamma'_v), (ac', av'), dst') \in E_h$, *such that :*

- $l_k = src$ *and* $l'_k = dst$ *and* $l_h = src'$ *and* $l'_h = dst'$ *and* $l'_i = l_i$ *for all* $i \neq h, k$,
- $eval((\gamma_c, \nu)) = eval((\gamma_v, A_k, v)) = eval((\gamma'_c, \nu)) = eval((\gamma'_v, A_h, v)) =$ *true,*
- $v' =$ *exec*$((av', A_h,$ *exec*$(av, A_k, v))))$ *and* $\nu' =$ *exec*$((\alpha'_c,$ *exec*$((\alpha_c, \nu))))$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l'_i), \nu')) =$ *true for all* $0 \leq i \leq n-1$, *and*
- *if there exists a* l_i *such that* $lt_i(l_i) =$ *committed, then* $lt_k(l_k) =$ *committed or* $lt_h(l_h) =$ *committed.*

Note that the exclamation mark side of the synchronization precedes the question mark side of the synchronization with respect to assignments. The last kind of transition is transition in which only time elapses.

Definition 3.18 (δ delay transition) *A tuple of states* $((\vec{l}, v, \nu), (\vec{l}, v, \nu'))$ *is a* δ *delay transition, where* $\delta \in \mathbb{R}^+$ *and* $\delta > 0$, *if*

- $\nu'(x) = \nu(x) + \delta$ *for all* $x \in X$,
- $eval((I_i(l_i), \nu)) = eval((I_i(l_i), \nu')) =$ *true for all* $0 \leq i \leq n-1$, *and*
- $lt_i(l_i) \neq$ *committed for all* $0 \leq i \leq n-1$,
- *if there exists an* i *such that* $lt_i(l_i) =$ *urgent, then no state* r *exists such that* $((\vec{l}, v, \nu), r)$ *is a simple action transition or* σ *action transition for some* $\sigma \in \Sigma$,
- *no state* r *exists such that* $((\vec{l}, v, \nu), r)$ *is a* σ *action transition for some* $\sigma \in \Sigma$ *such that* $ct(\sigma) =$ *urgent.*

With the definitions of states and the three transitions we have defined the structure of our transition system. We finish with describing the *initial state* of a SUPPAAL model. The location vector of this state is defined by the L_i^0 for every process i, the initial variable valuation is given by the *init* functions, and finally, the initial clock valuation assigns 0 to all clocks in the model.

Definition 3.19 (Run) *A finite or infinite sequence of states $s_0, s_1, ...$ is a run, if s_0 is the initial state, and (s_i, s_{i+1}) is a simple action transition, a σ action transition, or a δ delay transition for all i.*

For a SUPPAAL model M we let $\mathcal{R}(M)$ denote the set of all runs of M, which thus captures the behavior of M. Since the theory of symmetry that we adopt (summarized in section 3.2) is solely concerned with reachability of states, we limit ourselves to reachability properties. Let us assume that we have a set of *state properties* Φ, for whose elements ϕ we can easily say whether they are true or false in some state.

Definition 3.20 (Reachability) *For a SUPPAAL model M and a state property ϕ, we say that ϕ is reachable in M, denoted by $M \models \exists\Diamond\phi$, if a run $s_0, s_1, ... \in \mathcal{R}(M)$ exists such that ϕ is true in some s_i of that run.*

The model checking engine of UPPAAL can decide whether or not $M \models \exists\Diamond\phi$. It constructs a finite abstraction of the transition system of M on-the-fly, using *difference bounded matrices* for symbolic representation of the clock valuation of the state [84, 18, 43, 6]. This finite abstraction is then treated by a classical finite state model checking algorithm as depicted in Figure 3.1. It is not very difficult to adjust the engine to take symmetry into account, as is schematically depicted in Figure 3.2.

3.4 Extraction of Automorphisms

In this section we extract automorphisms from an SUPPAAL model which has been extended with scalarsets. First, we define the so-called swap functions, and second, we prove that these swap functions are automorphisms.

We assume the context of a SUPPAAL model $(\Omega, s, V^g, vt, init, X^g, \Sigma, ct, \mathbb{T})$ which gives rise to n (uniquely renamed!) processes, denoted by $A_i = (T_i, \rho_i)$.

3.4.1 Defining the automorphisms

As Ip and Dill we define permutations on the state graph of, in our case, a SUPPAAL model. Part of the state of a SUPPAAL model consists of local contributions of the various processes of the model. Moreover, the behavior of the model is defined by the control structure of the processes. Therefore, we use processes which are (almost) syntactically equivalent to permute the state.

Definition 3.21 (Process swap) *Let A_i and A_j be processes of M that originate from the same template ($T_i = T_j$). We define a process swap $\xi_{ij} : \mathbb{S} \to \mathbb{S}$ as follows: $\xi_{ij}((\vec{l}, v, \nu)) = (\vec{l'}, v', \nu')$, such that*

$$l'_i = l_j, l'_j = l_i, \text{and } l'_k = l_k \text{ for all } k \neq i, j$$

$$v'(a) = \begin{cases} v(\mathit{equiv}^V_{ij}(a)) & \text{if } a \in V_i \\ v(\mathit{equiv}^V_{ji}(a)) & \text{if } a \in V_j \\ v(a) & \text{otherwise} \end{cases}$$

$$\nu'(x) = \begin{cases} \nu(\mathit{equiv}^X_{ij}(x)) & \text{if } x \in X_i \\ \nu(\mathit{equiv}^X_{ji}(x)) & \text{if } x \in X_j \\ \nu(x) & \text{otherwise} \end{cases}$$

Lemma 3.22 *A process swap is its own inverse.*

PROOF. We prove that $\xi_{ij} \circ \xi_{ij}((\vec{l}, v, \nu)) = \xi_{ij}((\vec{l'}, v', \nu')) = (\vec{l''}, v'', \nu'')$ such that $\vec{l''} = \vec{l}$, $v'' = v$, and $\nu'' = \nu$. We split the proof in three parts:

- $\vec{l''} = \vec{l}$. From definition 3.21 we know that $l'_i = l_j$ and $l'_j = l_i$. Applying ξ_{ij} again gives us that $l''_i = l'_j$ and $l''_j = l'_i$. Thus, $l''_i = l_i$, and $l''_j = l_j$ and $l''_k = l_k$ for all $k \neq i, j$.
- $v'' = v$. We prove that $v''(a) = v(a)$ for all variables a in three cases:
  - $a \notin V_i \cup V_j$. We see in definition 3.21 that the value of a remains unchanged. Thus $v''(a) = v'(a) = v(a)$.
  - $a \in V_i$. In definition 3.21 we read that $v'(a) = v(\mathit{equiv}_{ij}(a))$. Since the equivalence function is a bijection from V_i to V_j, we know that $\mathit{equiv}_{ij}(a) \in V_j$. Applying another process swap thus gives us that
    $$v''(a) = v(\mathit{equiv}_{ji} \circ \mathit{equiv}_{ij}(a))$$
    At the end of section 3.3.2 we explained that $\mathit{equiv}_{ij} \circ \mathit{equiv}_{ji} = \mathbf{id}$. Therefore, we can say that $v''(a) = v(a)$.
  - $a \in V_j$. The proof of this case is similar as the proof in the previous item.
- $\nu'' = \nu$. We can proof this by an argument similar to the one in the previous item.

■

Next, we define the multiple process swap.

Definition 3.23 (Multiple process swap) *Let $\alpha \in \Omega$ be a scalarset and let $0 \leq i \neq j < s(\alpha)$. A multiple process swap is the composition $\xi^\alpha_{ij} = \xi_{k_1k_2} \circ \xi_{k_3k_4} \circ ... \circ \xi_{k_{2m-1}k_{2m}}$, such that for all processes $A_{k_1}, ..., A_{k_{2m}}$ the following holds:*

- $T_{k_{2p-1}} = T_{k_{2p}}$ *for all* $1 \leq p \leq m$,
- $\rho_{k_{2p-1}}(\alpha) = i$ *and* $\rho_{k_{2p}}(\alpha) = j$ *for all* $1 \leq p \leq m$,
- $\rho_{k_{2p-1}}(\beta) = \rho_{k_{2p}}(\beta)$ *for all* $1 \leq p \leq m$ *and for all* $\beta \neq \alpha$.

Moreover, there are no process indices k_{x-1} and k_x such that the three items above are satisfied, and $\xi_{k_{x-1}k_x}$ is not in the composition. (Note that $\xi_{ij} = \xi_{ji}$).

Lemma 3.24 *A process is swapped at most once by a multiple process swap.*

PROOF. Consider some process A_p such that $\rho_p(\alpha) = i$. Equation 3.3 allows us to conclude that there is *exactly* one other process A_q which originates from the same template, $\rho_q(\alpha) = j$, and whose other scalarsets match those of A_p. Therefore, only the process swap ξ_{pq} (or, equivalently, ξ_{qp}) involving A_p can be present in the multiple process swap. ■

Lemma 3.25 *If process A_p is not swapped by a multiple process swap ξ^{α}_{ij}, then $\rho_p(\alpha) \notin \{i, j\}$.*

PROOF. We prove that if $\rho_p(\alpha) \in \{i, j\}$, then the process is swapped, which is logically equivalent to our lemma. Thus, we assume $\rho_p(\alpha) = i$. Equation 3.3 allows us to conclude that there is *exactly* one other process A_q which originates from the same template, $\rho_q(\alpha) = j$, and whose other scalarsets match those of A_p. Therefore, the process swap ξ_{pq} must be in the multiple process swap ξ^{α}_{ij} according to definition 3.23. A similar argument holds for the situation $\rho_p(\alpha) = j$. ■

Lemma 3.26 *Consider four processes A_p, $A_{p'}$, A_q and $A_{q'}$ and two process swaps $\xi_{p,p'}$ and $\xi_{q,q'}$. If the four processes are all different, then $\xi_{p,p'} \circ \xi_{q,q'} = \xi_{q,q'} \circ \xi_{p,p'}$.*

PROOF. In definition 3.21 we see that the process swaps are orthogonal, because they only swap the *local* contributions to the state of the processes. Thus, if the four processes are different, then it does not matter in which order we apply the process swaps. ■

The second step swaps the dimensions of integer variable arrays which are indexed by some scalarset, and it swaps the integer variables which are target of an assignment with a scalarset constant.

Definition 3.27 (Data swap) *Let $\alpha \in \Omega$ be a scalarset and let $0 \leq i \neq j < s(\alpha)\}$. A data swap is defined as $\zeta^{\alpha}_{ij}((\vec{l}, v, \nu)) = (\vec{l}, v', \nu)$, such that: for every regular variable a:*

$$v'(a) = \begin{cases} i & \text{if } v(a) = j \text{ and } a \in \text{used}_{\alpha} \\ j & \text{if } v(a) = i \text{ and } a \in \text{used}_{\alpha} \\ v(a) & \text{otherwise} \end{cases}$$

And for every n dimensional integer array a:

$$v'(a[i_0]...[i_{n-1}]) = v(a[(i_0)_{\alpha,0}]...[(i_{n-1})_{\alpha,n-1}])$$

where the functions $()_{\alpha,k} : \mathbb{N} \to \mathbb{N}$ are defined for array a as:

$$(c)_{\alpha,k} = \begin{cases} i & \text{if } c = j \text{ and } [tVar(a)]_k = \alpha \\ j & \text{if } c = i \text{ and } [tVar(a)]_k = \alpha \\ c & \text{otherwise} \end{cases}$$

Lemma 3.28 *A data swap is its own inverse.*

PROOF. We prove that $\zeta^\alpha_{ij} \circ \zeta^\alpha_{ij}((\vec{l}, v, \nu)) = \zeta^\alpha_{ij}((\vec{l'}, v', \nu')) = (\vec{l''}, v'', \nu'')$ such that $\vec{l''} = \vec{l}$, $v'' = v$, and $\nu'' = \nu$. We split the proof in three parts:

- $\vec{l''} = \vec{l}$. From definition 3.27 we know that the data swap does not alter the location vector. Therefore, $\vec{l''} = \vec{l}$.
- $v'' = v$. First, we proof that $v''(a[i_0]...[i_{n-1}]) = v(a[i_0]...[i_{n-1}])$ for all n dimensional integer arrays a. Therefore, let us apply two data swaps to this array:

  $$v''(a[i_0]...[i_{n-1}]) = v(a[((i_0)_{\alpha,0})_{\alpha,0}]...[((i_{n-1})_{\alpha,n-1})_{\alpha,n-1}])$$

  Next, we prove that $((c)_{\alpha,k})_{\alpha,k} = c$. We distinguish three cases. First, $c = i$ and $[tVar(a)]_k = \alpha$. By definition $((c)_{\alpha,k})_{\alpha,k} = (j)_{\alpha,k} = i$. Second, $c = j$ and $[tVar(a)]_k = \alpha$. Then $((c)_{\alpha,k})_{\alpha,k} = (i)_{\alpha,k} = j$. The third case encompasses all other situations, and thus $((c)_{\alpha,k})_{\alpha,k} = (c)_{\alpha,k} = c$ by definition 3.27.

  Second, we must prove that $v''(a) = v(a)$ for all regular variables a. Again, we distinguish three cases. First, $v(a) = j$ and $a \in used_\alpha$. By definition $v'(a) = i$ and – of course – still $a \in used_\alpha$. Applying another swap thus gives us by definition that $v''(a) = j$. The second case, $v(a) = i$ and $a \in used_\alpha$, is similar. The third case encompasses all other situations, and again by definition 3.27 we know that $v''(a) = v'(a) = v(a)$.
- $\nu'' = \nu$, since the data swap does not alter the clock valuation.

■

Lemma 3.29 *Consider a process swap ξ_{pq} and a data swap ζ^α_{ij}. The order of application does not matter: $\xi_{pq} \circ \zeta^\alpha_{ij} = \zeta^\alpha_{ij} \circ \xi_{pq}$.*

PROOF. Assume that $\xi_{pq} \circ \zeta^\alpha_{ij}((\vec{l}, v, \nu)) = \xi_{pq}((\vec{l}_1, v'_1, \nu'_1)) = (\vec{l}''_1, v''_1, \nu''_1)$, and similarly $\zeta^\alpha_{ij} \circ \xi_{pq}((\vec{l}, v, \nu)) = \zeta^\alpha_{ij}((\vec{l}_2, v'_2, \nu'_2)) = (\vec{l}''_2, v''_2, \nu''_2)$. We prove that $(\vec{l}''_1, v''_1, \nu''_1) = (\vec{l}''_2, v''_2, \nu''_2)$.

- $\vec{l}_1'' = \vec{l}_2''$, because we now by definition 3.27 that the data swap does not alter the location vector.
- $v_1'' = v_2''$, thus $v_1''(a) = v_2''(a)$ for all (arrays of) integer variables a. We distinguish three cases:
  - $a \in V_p$. There are two situations. First, a can be a regular (non-array) variable. Then $v_1'(a)$ equals i if $v(a) = j$ and $a \in \textit{used}_\alpha$, or j if $v(a) = i$ and $a \in \textit{used}_\alpha$, or $v(a)$ otherwise. Similarly, $v_1'(b)$ equals i if $v(b) = j$ and $b \in \textit{used}_\alpha$, or j if $v(b) = i$ and $b \in \textit{used}_\alpha$, or $v(b)$ otherwise, where $b = \textit{equiv}_{pq}(a)$. Note that b exists since a is a local variable of process A_p. Applying the process swap swaps the value of a with the value of b, thus:

    $$v_1''(a) = v_1'(b) = \begin{cases} i & \text{if } v(b) = j \text{ and } b \in \textit{used}_\alpha \\ j & \text{if } v(b) = i \text{ and } b \in \textit{used}_\alpha \\ v(b) & \text{otherwise} \end{cases}$$

    Next, we consider $v_2'(a)$, which results from a process swap. Thus: $v_2'(a) = v(b)$, where $b = \textit{equiv}_{pq}(a)$. Applying the data swap gives us by definition:

    $$v_2''(a) = \begin{cases} i & \text{if } v_2'(a) = v(b) = j \text{ and } b \in \textit{used}_\alpha \\ j & \text{if } v_2'(a) = v(b) = i \text{ and } b \in \textit{used}_\alpha \\ v(b) & \text{otherwise} \end{cases}$$

    Therefore, $v_1''(a) = v_2''(a)$ for all regular variables a.
    Second, a can be a n-dimensional array of integers. Applying the data swap first gives $v_1'(a[i_0]...[i_{n-1}]) = v(a[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$. The process swap then swaps all entries of a with $\textit{equiv}_{pq}(a)$:

    $$v_1''(a[i_0]...[i_{n-1}]) = v(\textit{equiv}_{pq}(a)[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$$

    On the other hand, applying the process swap first gives us that $v_2'(a[i_0]...[i_{n-1}]) = v(\textit{equiv}_{pq}(a)[i_0]...[i_{n-1}])$. Next, we apply the data swap with the result that

    $$v_2''(a[i_0]...[i_{n-1}]) = v(\textit{equiv}_{pq}(a)[(i_0)_{\alpha,0}]...[(i_{n-1}])_{\alpha,n-1})$$

    Thus, we conclude that $v_1''(a) = v_2''(a)$ for all (arrays of) integer variables a.
  - $a \in V_q$. We can prove this with a similar argument as appears in the previous item.
  - $a \notin V_p \cup V_q$. These variables are left unchanged by the process swap, as we can read in definition 3.21. Therefore, obviously $v_1''(a) = v_2''(a)$ for these variables.

- $\nu_1'' = \nu_2''$, because we now by definition 3.27 that the data swap does not alter the clock valuation.

■

Using the multiple process swap and the data swap we can define the permutations which permute the i-th and the j-th element of a scalarset consistently through the state.

Definition 3.30 (State swap) *Let α be a scalarset and let $0 \leq i \neq j < s(\alpha)$. Then, $\zeta_{ij}^{\alpha} \circ \xi_{ij}^{\alpha}$ is a state swap abbreviated by π_{ij}^{α}.*

Lemma 3.31 *A state swap is its own inverse.*

PROOF. Consider a state swap $p_1 \equiv \zeta_{ij}^{\alpha} \circ \xi_{ij}^{\alpha} = \zeta_{ij}^{\alpha} \circ \xi_{k_1k_2} \circ \xi_{k_3k_4} \circ ... \circ \xi_{k_{2m-1}k_{2m}}$. Since a multiple process swap does always swap a process at most once (see lemma 3.24), we can use lemma 3.26 and lemma 3.29 to rewrite this state swap to $p2 \equiv \xi_{k_{2m-1}k_{2m}} \circ ... \circ \xi_{k_1k_2} \circ \zeta_{ij}^{\alpha}$. Thus $p_1 = p_2$, and using lemma 3.22 and lemma 3.28 we can rewrite $p_1 \circ p_2$ in the following way:

$$\begin{aligned} p_1 \circ p_2 &= \zeta_{ij}^{\alpha} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-1}k_{2m}} \circ \xi_{k_{2m-1}k_{2m}} \circ ... \circ \xi_{k_1k_2} \circ \zeta_{ij}^{\alpha} \\ &= \zeta_{ij}^{\alpha} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-3}k_{2m-2}} \circ \mathbf{id} \circ \xi_{k_{2m-3}k_{2m-2}} \circ ... \circ \xi_{k_1k_2} \circ \zeta_{ij}^{\alpha} \\ &= \zeta_{ij}^{\alpha} \circ \xi_{k_1k_2} \circ ... \circ \xi_{k_{2m-3}k_{2m-2}} \circ \xi_{k_{2m-3}k_{2m-2}} \circ ... \circ \xi_{k_1k_2} \circ \zeta_{ij}^{\alpha} \\ &= .... \\ &= \mathbf{id} \end{aligned}$$

Thus, a state swap is its own inverse. ■

In the next section we define swap functions on the *syntax* of our models. We use these to prove that the state swaps as defined above are automorphisms.

3.4.2 Syntactical Swaps

This section defines swaps on the syntax of a SUPPAAL model. We use these swaps in the soundness proof in the next section. In the remainder of this section we abbreviate the state swap π_{ij}^{α} by π, while maintaining the parameters α, i and j.

Definition 3.32 (Syntactical integer expression swap) *A syntactical integer expression swap is a function $\pi_{pq}^{s} : IX(V^g \cup V_p, S_p) \rightarrow IX(V^g \cup V_q, S_q)$ defined as*

follows:

$$\pi^s_{pq}(exp) = \begin{cases} exp & \text{if } exp \in \mathbb{Z} \cup S_p \\ exp & \text{if } exp \in V \setminus V_p \\ \textit{equiv}_{pq}(exp) & \text{if } exp \in V_p \\ v[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] & \text{if } exp \equiv v[e_1]...[e_m] \\ & \text{and } v \in V \setminus V_p \\ \textit{equiv}_{pq}(v)[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] & \text{if } exp \equiv v[e_1]...[e_m] \\ & \text{and } v \in V_p \\ (\pi^s_{pq}(e_1) \oslash \pi^s_{pq}(e_2)) & \text{if } exp \equiv (e_1 \oslash e_2) \end{cases}$$

Lemma 3.33 *Let $e \in IX(V^g \cup V_p, S_p)$ be a well-formed expression of process A_p, let π be a state swap and let π^s_{pq} be a syntactical integer expression swap. If $e \neq \alpha$ and $e \notin \textit{used}_\alpha$, then*

(1) If π swaps A_p with A_q, then it is true that $\textit{eval}(e, A_p, [(\vec{l}, v, \nu)]_1)$ equals $\textit{eval}(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1)$.

(2) If π does not swap A_p, then it is true that $\textit{eval}(e, A_p, [(\vec{l}, v, \nu)]_1)$ equals $\textit{eval}(e, A_p, [\pi(\vec{l}, v, \nu)]_1)$.

PROOF. We proof all seven cases of definition 3.32 separately for both parts of the lemma.

- $e \in \mathbb{Z}$. We know that $\textit{eval}(e, A_p, [(\vec{l}, v, \nu)_1]) = z$ by definition of the evaluation function. And $\textit{eval}(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) = z$, since no variable interpretation is needed for the evaluation of a number, and the syntactical swap does not change such a number by definition. This proves part (1) and part (2) of the lemma.

- $e = \beta$ for a scalarset $\beta \neq \alpha$. We know by definition of the evaluation function that $\textit{eval}(e, A_p, [(\vec{l}, v, \nu)]_1) = \rho_p(e)$. And therefore:

  $$\textit{eval}(\pi^s_{pq}(e), [\pi(\vec{l}, v, \nu)]_1) = \textit{eval}(e, [\pi(\vec{l}, v, \nu)]_1) = \rho_q(e)$$

  In definition 3.23 we read that A_p and A_q do only differ in the value of their α scalarset. Therefore, $\rho_p(e) = \rho_q(e)$. This proves part (1) of the lemma. As for part (2), we say that the variable interpretation is not needed to evaluate scalarset e.

- $e \in V \setminus V_p$. Obviously, $\textit{eval}(e, A_p, [(\vec{l}, v, \nu)_1]) = v(e)$. Second,

  $$\textit{eval}(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) = \textit{eval}(e, A_q, [\pi(\vec{l}, v, \nu)]_1) = \textit{eval}(e, A_p, v') = v'(e)$$

  By definition, π's process swaps do not alter e, since e is not local to A_p. Moreover, π's data swap does also not alter the value of e, since we assumed in the lemma that $e \notin \textit{used}_\alpha$. Therefore, $v'(e) = v(e)$. This proves part (1) of the lemma, and with a very similar argument we can easily prove part (2).

- $e \in V_p$. Obviously, $\mathit{eval}(e, A_p, [(\vec{l}, v, \nu)_1]) = v(e)$. Second,

$$\begin{aligned} \mathit{eval}(\pi^s_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) &= \mathit{eval}(\mathit{equiv}_{pq}(e), A_q, [\pi(\vec{l}, v, \nu)]_1) \\ &= \mathit{eval}(\mathit{equiv}_{pq}(e), A_q, v') \end{aligned}$$

  By definition, π's process swap does alter the value of e in the following way: $v'(e) = v(\mathit{equiv}_{pq}(e))$. Moreover, $v'(\mathit{equiv}_{pq}(e)) = v(\mathit{equiv}_{qp} \circ \mathit{equiv}_{pq}(e)) = v(e)$, since $\mathit{equiv}_{qp} \circ \mathit{equiv}_{pq} = \mathbf{id}$. The data swap does not alter the value of e, since we assumed in the lemma that $e \notin \mathit{used}_\alpha$. Thus, $\mathit{eval}(\mathit{equiv}_{pq}(e), A_q, v')$ equals $v(e)$. This proves part (1) of the lemma. As for part (2), note that A_p is not swapped. Therefore, the value of the local variable e is not changed.

- $e \equiv (e_1 \oslash e_2)$ By definition 3.6 we know that e_1 and e_2 are simple integer expressions. For part (1) of the lemma we must prove that

$$\mathit{eval}((e_1 \oslash e_2), A_p, [(\vec{l}, v, \nu)]_1) = \mathit{eval}(\pi^s_{pq}(e_1 \oslash e_2), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

  And thus by definition 3.32:

$$\begin{gathered} \mathit{eval}(e_1, A_p, [(\vec{l}, v, \nu)]_1) \oslash \mathit{eval}(e_2, A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ \mathit{eval}(\pi^s_{pq}(e_1), A_q, [\pi(\vec{l}, v, \nu)]_1) \oslash \mathit{eval}(\pi^s_{pq}(e_2), A_q, [\pi(\vec{l}, v, \nu)]_1) \end{gathered}$$

  This can easily be proved by induction on the syntax of e_1 and e_2. The base is formed by the four previous items. The induction step is straightforward and we do not explicitly explain it. The proof of part (2) is very similar.

- $exp \equiv a[e_0]...[e_m]$ and $a \in V \setminus V_p$. We start with the proof of part (1) of the lemma. First we rewrite the first term in our lemma as follows:

$$\mathit{eval}(a[e_0]...[e_m], A_p, v) = v(a[\mathit{eval}(e_0, A_p, v)]...[\mathit{eval}(e_m, A_p, v)])$$

  Second, we rewrite the second term in our lemma using the definition of the evaluation function and definition 3.32:

$$\begin{gathered} \mathit{eval}(\pi^s_{pq}(a[e_0]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) = \\ \mathit{eval}(a[\pi^s_{pq}(e_0)]...[\pi^s_{pq}(e_m)], A_q, v') = \\ v'(a[\mathit{eval}(\pi^s_{pq}(e_0), A_q, v')]...[\mathit{eval}(\pi^s_{pq}(e_m), A_q, v')]) \end{gathered}$$

  where $v' \equiv [\pi(\vec{l}, v, \nu)]_1$. Note that π does not "alter" the values in the array a, since the process swap of π does not affect a (since $a \notin V_p$). Thus, entries of a are merely swapped around (see definition 3.27 of the data swap). Therefore, we can rewrite the previous term to:

$$v(a[(\mathit{eval}(\pi^s_{pq}(e_0), A_q, v'))_{\alpha,0}]...[(\mathit{eval}(\pi^s_{pq}(e_m), A_q, v'))_{\alpha,m}])$$

Thus, we must prove the following equality for all array dimensions k:

$$\mathit{eval}(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

Again, we must consider all possible cases for the syntax of e_k. In definition 3.6 we see that e_k must be a simple integer expression. Thus, we enumerate the possibilities:

- $e_k \in \mathbb{Z}$. We know by definition that $\mathit{eval}(e_k, A_p, v) = e_k$, and also that $(\mathit{eval}(e_k, A_q, v'))_{\alpha,k} = (e_k)_{\alpha,k}$. Since we assumed that our expression is well-formed, this k-th dimension may not be a scalarset dimension. Therefore, $(e_k)_{\alpha,k} = e_k$.
- $e_k = \beta$ for some scalarset β. By definition $\mathit{eval}(e_k, A_p, v) = \rho_p(e_k)$. As for the second part:

  $$(\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = (\rho_q(e_k))_{\alpha,k}$$

  We now can distinguish two cases. First, $\beta \neq \alpha$. Since the expression is well-formed, we know that the k-th dimension is not an α dimension. Thus we conclude that $(\rho_q(e_k))_{\alpha,k} = \rho_q(e_k)$. Since π swaps process A_p with A_q, we can conclude from definition 3.23 that the β constant of these processes is the same. Thus: $\rho_p(e_k) = \rho_q(e_k)$.
  Second, $\beta = \alpha$. Since π swaps process A_p with A_q, we know that $\rho_p(e_k) = i$, and $\rho_q(e_k) = j$. Thus, $(\rho_q(e_k))_{\alpha,k} = (j)_{\alpha,k} = i$ by definition, because the k-th dimension now is an α dimension.
- $e_k \in V^g \cup V_p$ or $e_k \equiv (e_1 \oslash e_2)$, where e_1 and e_2 are simple expressions. Since we assumed that the expression under consideration is well-formed, we conclude that dimension k is a non-scalarset dimension. Therefore,

  $$(\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = \mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

  The proof for this case can be found in the third, fourth and fifth main item of the proof of this lemma.

Next, we prove the second part of the lemma. With an argument similar to the one at the start of this item, we conclude that we must prove the equality

$$\mathit{eval}(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

where e_k is a simple integer expression. Again, we distinguish three cases.

- $e_k \in \mathbb{Z}$. We know by definition that $\mathit{eval}(e_k, A_p, v) = e_k$, and also that $(\mathit{eval}(e_k, A_p, v'))_{\alpha,k} = (e_k)_{\alpha,k}$. Since we assumed that our expression is well-formed, this k-th dimension may not be a scalarset dimension. Therefore, $(e_k)_{\alpha,k} = e_k$.

- $e_k = \beta$ for some scalarset β. By definition $\mathit{eval}(e_k, A_p, v) = \rho_p(e_k)$. As for the second part:

$$(\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = (\rho_p(e_k))_{\alpha,k}$$

  We now can distinguish two cases. First, $\beta \neq \alpha$. Since the expression is well-formed, we know that the k-th dimension is not an α dimension. Thus, $(\rho_p(e_k))_{\alpha,k} = \rho_p(e_k)$. Second, $\beta = \alpha$. Since π does not swap process A_p, we know by lemma 3.25 that $\rho_p(e_k) \neq i, j$. Thus, $(\rho_p(e_k))_{\alpha,k} = \rho_p(e_k)$.

- $e_k \in V^g \cup V_p$ or $e_k \equiv (e_1 \oslash e_2)$, where e_1 and e_2 are simple expressions. Since we assumed that the expression under consideration is well-formed, we conclude that dimension k is a non-scalarset dimension. Therefore,

$$(\mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k} = \mathit{eval}(e_k, A_p, [\pi(\vec{l}, v, \nu)]_1)$$

  The proof for this case can be found in the third, fourth and fifth main item of the proof of this lemma.

- $exp \equiv a[e_0]...[e_m]$ and $a \in V_p$. We start with the first part of the lemma. We rewrite the first term in our lemma as follows:

$$\mathit{eval}(a[e_0]...[e_m], A_p, v) = v(a[\mathit{eval}(e_0, A_p, v)]...[\mathit{eval}(e_m, A_p, v)])$$

  Second, we rewrite the second term in our lemma using the definition of the evaluation function and definition 3.32:

$$\begin{array}{l}\mathit{eval}(\pi^s_{pq}(a[e_0]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) = \\ \mathit{eval}(\mathit{equiv}_{pq}(a)[\pi^s_{pq}(e_0)]...[\pi^s_{pq}(e_m)], A_q, v')\end{array}$$

  Again, we can rewrite this to

$$v'(\mathit{equiv}_{pq}(a)[\mathit{eval}(\pi^s_{pq}(e_0), A_q, v')]...[\mathit{eval}(\pi^s_{pq}(e_m), A_q, v')])$$

  Note that π consists of a multiple process swap and a data swap. The process swaps swap entire arrays, while the data swap swaps dimensions of arrays. More precisely, $v'(\mathit{equiv}_{pq}(a)[i_0]...[i_m])$ is equal to $v(\mathit{equiv}_{qp} \circ \mathit{equiv}_{pq}(a)[(i_0)_{\alpha,0}]...[(i_m)_{\alpha,m}])$. Thus, we can rewrite the previous term to:

$$v(a[(\mathit{eval}(\pi^s_{pq}(e_0), A_q, v'))_{\alpha,0}]...[(\mathit{eval}(\pi^s_{pq}(e_m), A_q, v'))_{\alpha,m}])$$

  Thus, we must prove the following equality for all array dimensions k:

$$\mathit{eval}(e_k, A_p, [(\vec{l}, v, \nu)]_1) = (\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

  We have already done this in the previous item.

  The proof of the second part of the lemma is very similar to the proof in the previous item, and we do not explicitly explain it.

■

Lemma 3.34 *Let us consider some (β, n)-malformed integer expression in dimension d, say $a[e_0]...[e_{d-1}][n][e_{d+1}]...[e_m]$. There exists an $n' \in \{0, ..., |\beta| - 1\}$ such that*

$$\begin{gathered} \mathit{eval}(a[e_0]...[e_{d-1}][n][e_{d+1}]...[e_m], A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ \mathit{eval}(\pi^s_{pq}(a[e_0]...[e_{d-1}][n'][e_{d+1}]...[e_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \end{gathered}$$

PROOF. We can use the same argument as in the last two items of the proof of lemma 3.33 to argue that we first must prove that

$$\mathit{eval}(e_k, A_p, [(\vec{l}, v, \nu)]_1 = (\mathit{eval}(\pi^s_{pq}(e_k), A_q, [\pi(\vec{l}, v, \nu)]_1))_{\alpha,k}$$

for every array dimension $k \neq d$. We can use the last two items of the proof of lemma 3.33 to prove this.

Now we consider the remaining situation for dimension d. Since the expression is (β, n)-malformed in this dimension, we thus know that $n \in \{0, ..., |\beta| - 1\}$. From definition 3.32 it is clear that $\pi^s_{pq}(n) = n$. Moreover, we do not need the variable interpretation for the evaluation of n. Thus, we must show that we can find an $n' \in \{0, ..., |\beta| - 1\}$ such that:

$$n = (n')_{\alpha,d}$$

It is not difficult to see that the following definition of n' satisfies this equality:

$$n' = \begin{cases} i & \text{if } n = j \text{ and } \alpha = \beta \\ j & \text{if } n = i \text{ and } \alpha = \beta \\ n & \text{otherwise} \end{cases}$$

(See definition 3.27 for the definition of the $()_{\alpha,k}$ function.) ■

We can also define syntactical swaps of the integer assignments. These functions take an integer assignment of process A_p and change the syntax in such a way that it becomes an integer assignment of process A_q, if A_p and A_q originate from the same template.

Definition 3.35 (Syntactical integer assignment swap) *A swap of a syntactical integer assignment is a function $\pi^s_{pq} : IA(V^g \cup V_p, S_p) \rightarrow IA(V^g \cup V_q, S_q)$ defined as follows:*

$$\pi^s_{pq}(a) = \begin{cases} b := \pi^s_{pq}(exp) & \textit{if } a \equiv b := exp \\ & \textit{and } b \notin V_p \\ \mathit{equiv}_{pq}(b) := \pi^s_{pq}(exp) & \textit{if } a \equiv b := exp \\ & \textit{and } b \in V_p \\ b[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] := \pi^s_{pq}(exp) & \textit{if } a \equiv b[e_1]...[e_m] \\ & \textit{:=exp and } b \notin V_p \\ \mathit{equiv}_{pq}(b)[\pi^s_{pq}(e_1)]...[\pi^s_{pq}(e_m)] := \pi^s_{pq}(exp) & \textit{if } a \equiv b[e_1]...[e_m] \\ & \textit{:=exp and } b \in V_p \end{cases}$$

where the function π^s_{pq} which appears right of the large bracket is the syntactical integer expression swap.

The next lemma states that an integer assignment in process A_p has the same effect as an integer assignment in process A_q modulo symmetry, if they are swapped by a state swap.

Lemma 3.36 *Consider a well-formed integer assignment $ia \in IA(V^g \cup V_p, S_p)$ of A_p, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $v' = exec((ia, A_p, v))$, a syntactical assignment swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then it is true that $exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1))$ equals $[\pi(\vec{l'}, v', \nu')]_1$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_1 = exec(ia, A_p, [\pi(\vec{l}, v, \nu)]_1))$.

PROOF. We use the following abbreviations in our proof: $v'' = [\pi(\vec{l'}, v', \nu')]_1$ and $v''' = exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1))$. We start with part (1) of the lemma, for which we distinguish two cases. First, ia is of the form $a := exp$. In this case, we distinguish another four cases:

- $a \in V_p$ and $a \in \mathit{used}_\beta$ for some scalarset β. In this case, exp may only be equivalent to to a value $z \in \mathbb{Z} \setminus \{0, 1, ..., s(\beta) - 1\}$ or to β (see restriction (2c) on page 43). In the first situation,

$$\begin{gathered} v''' = exec((\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1)) = \\ exec((\mathit{equiv}_{pq}(a) := \pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)) = \\ exec((\mathit{equiv}_{pq}(a) := z, A_q, [\pi(\vec{l}, v, \nu)]_1)) \end{gathered}$$

  Thus, we can define v''' as follows:

$$v'''(b) = \begin{cases} z & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

  Similarly, we can deduct that

$$v'(b) = \begin{cases} z & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$

  And therefore we can conclude that

$$v''(b) = [\pi(\vec{l'}, v', \nu')]_1(b) = \begin{cases} z & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

  Concluding, $v'' = v'''$. In the second case we can rewrite v''' as follows:

$$\begin{aligned} v''' &= exec((\pi^s_{pq}(a := \beta), A_q, [\pi(\vec{l}, v, \nu)]_1)) \\ &= exec((\mathit{equiv}_{pq}(a) := \beta, A_q, [\pi(\vec{l}, v, \nu)]_1)) \end{aligned}$$

Thus, we can define v''' as follows:

$$v'''(b) = \begin{cases} \rho_q(\beta) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Similarly, we can deduct that

$$v'(b) = \begin{cases} \rho_p(\beta) & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$

Now we distinguish two further possibilities. First, $\beta = \alpha$. Then we now by definition 3.23 that $\rho_p(\beta) = i$, and that $\rho_q(\beta) = j$. Applying the state swap thus changes the value of a and swaps it with its equivalent in process A_q:

$$v''(b) = [\pi(\vec{l}', v', \nu')]_1(b) = \begin{cases} j & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

We can conclude that $v'' = v'''$. Second, $\beta \neq \alpha$. In this case we now by definition 3.23 that $\rho_p(\beta) = \rho_q(\beta)$. Applying the state swap to v' does only swap the variable a with its equivalent in process A_q:

$$v''(b) = [\pi(\vec{l}', v', \nu')]_1(b) = \begin{cases} \rho_p(\beta) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Again, we can conclude that $v'' = v'''$.

- $a \in V^g$ and $a \in \mathit{used}_\beta$ for some scalarset β. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

- $a \in V_p$ and $a \notin \mathit{used}_\beta$ for all scalarsets β. Again, we rewrite v''' using the definitions.

$$\begin{gathered} v''' = exec((\pi^s_{pq}(a := exp), A_q, [\pi(\vec{l}, v, \nu)]_1)) \\ = \\ exec((\mathit{equiv}_{pq}(a) := \pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)) \end{gathered}$$

Thus, we can define v''' as follows:

$$v'''(b) = \begin{cases} \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Similarly, we can deduct that

$$v'(b) = \begin{cases} \mathit{eval}(exp, A_p, v) & \text{if } b = a \\ v(b) & \text{otherwise} \end{cases}$$

And therefore we can conclude that

$$v''(b) = [\pi(\vec{l}', v', \nu')]_1(b) = \begin{cases} \mathit{eval}(exp, A_p, v) & \text{if } b = \mathit{equiv}_{pq}(a) \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

To prove $v'' = v'''$ we prove that $\mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}(exp, A_p, v)$. We know that $exp \neq \beta$ for all scalarsets β, since $a \notin \mathit{used}_\beta$. Moreover, according to restriction (3) on page 43, $exp \notin \mathit{used}_\beta$ for all scalarsets β. Therefore, we can immediately use part (1) of lemma 3.33 to conclude that $v'' = v'''$.

- $a \in V^g$ and $a \notin \mathit{used}_\beta$ for all scalarsets β. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

Second, ai is of the form $a[i_0]...[i_m] := exp$. By the restrictions on the syntax of the model stated on page 42, we know that $a, exp \notin \mathit{used}_\beta$ and $exp \neq \beta$ for all scalarsets β. We only distinguish two other cases:

- $a \in V_p$. We can rewrite v''' as follows:

$$v''' = \mathit{exec}(\pi^s_{pq}(a[i_0]...[i_m] := exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$$
$$=$$
$$\mathit{exec}(\mathit{equiv}_{pq}(a)[\pi^s_{pq}(i_0)]...[\pi^s_{pq}(i_m)] := \pi^s_{pq}(exp)), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

This enables us to define v''' as follows:

$$v'''(b) = \begin{cases} \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1) & \text{if } b = \mathit{equiv}_{pq}(a)[i'_0]...[i'_m] \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

where $i'_k = \mathit{eval}(\pi^s_{pq}(i_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Next, we construct a definition for v'' using the following definition of v':

$$v'(b) = \begin{cases} \mathit{eval}(exp, A_p, v) & \text{if } b = a[\mathit{eval}(i_0, A_p, v)]...[\mathit{eval}(i_m, A_p, v)] \\ v(b) & \text{otherwise} \end{cases}$$

The process swap of π interchanges the whole array with an equivalent array in process A_q, and it swaps around the data in the array:

$$v''(b) = \begin{cases} \mathit{eval}(exp, A_p, v) & \text{if } b = \mathit{equiv}_{pq}(a)[(\mathit{eval}(i_0, A_p, v))_{\alpha,0}]... \\ & \quad [(\mathit{eval}(i_m, A_p, v))_{\alpha,m}] \\ [\pi(\vec{l}, v, \nu)]_1(b) & \text{otherwise} \end{cases}$$

Thus, we must prove the following two statements in order to prove that $v'' = v'''$:

  - $\mathit{eval}(exp, A_p, v) = \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Due to the form of exp, we can immediately use part (1) of lemma 3.33 to conclude that this statement is true.

– $(\mathit{eval}(i_k, A_p, v))_{\alpha,k} = \mathit{eval}(\pi^s_{pq}(i_k), A_q, [\pi(\vec{l}, v, \nu)]_1)$ for all simple integer expressions i_k. There are many possibilities for the form of i_k. If $i_k \neq \alpha$ and $i_k \notin \mathit{used}_\alpha$, then we can immediately use part (1) of lemma 3.33 in conjunction with the fact that the array assignment is well formed to conclude that this statement is true.

Now let us consider the other cases. First, assume that $i_k = \alpha$. Since A_p is swapped, we know by definition 3.23 that $\rho_p(i_k) = i$, and that $\rho_q(\mathit{equiv}_{pq}(i_k)) = j$. Applying the definition of the $()_{\alpha,k}$ function gives us that $(\mathit{eval}(i_k, v))_{\alpha,k} = j$. Now, $\mathit{eval}(\pi^s_{pq}(i_k), [\pi(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}(\mathit{equiv}_{pq}(i_k), [\pi(\vec{l}, v, \nu)]_1)$, which equals $\rho_q(\mathit{equiv}_{pq}(i_k))$. This proves the first case. The second case, $i_k \in \mathit{used}_\alpha$, cannot exist, since variables in the *used* sets may not be used to index arrays according to the restrictions on page 42.

- $a \in V^g$. We can proof this item with a proof very similar to the one in the previous item. The only difference is that the multiple process swap of π now has no effect on the value of a.

This concludes part (1) of the lemma. As for part (2) we only say that the proof is very similar – with respect to the structure – to the proof given above. The key observation is that the state swap π now does not affect local variables of a, and that $\rho_p(\alpha) \neq i, j$ by lemma 3.25.

■

Lemma 3.37 *Consider a (β, n)-malformed integer assignment $ia \in IA(V^g \cup V_p, S_p)$, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $v' = exec((ia, A_p, v))$, a syntactical assignment swap π^s_{pq} and a state swap π. We can find a $n' \in \{0, ..., |\beta|-1\}$ such that for ia', which results from replacing malformation n by n' in ia, the following holds:*

(1) If π swaps A_p with A_q, then it is true that $exec((\pi^s_{pq}(ia'), A_q, [\pi(\vec{l}, v, \nu)]_1))$ equals $[\pi(\vec{l'}, v', \nu')]_1$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_1 = exec(ia', A_p, [\pi(\vec{l}, v, \nu)]_1))$.

PROOF. We can follow the structure of the proof of the previous lemma. This gives the desired result without much effort, when used in conjunction with lemma 3.34.
■

Lemma 3.38 *Consider a well-formed integer guard $g \in IG(V^g \cup V_p, S_p)$ of A_p, a state $(\vec{l}, v, \nu)$, a syntactical assignment swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then it is true that $\mathit{eval}(g, A_p, [(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}((\pi^s_{pq}(g), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then it is true that eval$(g, A_p, [(\vec{l}, v, \nu)]_1)$ equals eval$((g, A_p, [\pi(\vec{l}, v, \nu)]_1))$.

PROOF. We start by proving part (1) of the lemma. According to definition 3.7, the integer guard g has the form $e_1 \sim e_2$, where $e_1, e_2 \in IX(V^g \cup V_p, S_p)$. By definition, the evaluation of the integer guard is expressed by the evaluation of both integer expressions. We prove that

$$\textit{eval}((e_1 \sim e_2), A_p, [(\vec{l}, v, \nu)]_1) = \textit{eval}(\pi^s_{pq}(e_1 \sim e_2), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

Which is, by definition, equivalent to the following equality:

$$\begin{gathered}\textit{eval}(e_1, A_p, [(\vec{l}, v, \nu)]_1) \sim \textit{eval}(e_2, A_p, [(\vec{l}, v, \nu)]_1) \\ = \\ \textit{eval}(\pi^s_{pq}(e_1), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim \textit{eval}(\pi^s_{pq}(e_2), A_q, [\pi(\vec{l}, v, \nu)]_1)\end{gathered}$$

This can easily be proved for a large set of integer expressions using lemma 3.33. However, the following, remaining, cases are not yet covered by using the lemma:

- $g \equiv a = \alpha$ or $g \equiv a \neq \alpha$, where $a \in \textit{used}_\alpha$ and $\alpha \in S_p$. If $a \in V^g$, then we can rewrite the equality above to:

  $$v(a) \sim \rho_p(\alpha) = \textit{eval}(a, [\pi(\vec{l}, v, \nu)]_1) \sim \rho_q(\alpha)$$

  We now by definition of the state swap that $\rho_p(\alpha) = i$ and $\rho_q(\alpha) = j$. Now we can distinguish three cases. First, $v(a) = i$. Since $a \in \textit{used}_\alpha$, the data swap of π is such that $[\pi(\vec{l}, v, \nu)]_1(a) = j$. This proves the first case. The second case, $v(a) = j$, can be proved with a similar argument. Third, $v(a) \neq i, j$. Now, the data swap does not alter the value of a and we can derive: $v(a) \sim i = [\pi(\vec{l}, v, \nu)]_1(a) \sim j$ for $\sim \in \{=, \neq\}$, since both variable interpretations assign a value not equal to i or j to a.

  Now the case remains where $a \in V_p$. We must prove:

  $$v(a) \sim \rho_p(\alpha) = \textit{eval}(\textit{equiv}_{pq}(a), [\pi(\vec{l}, v, \nu)]_1) \sim \rho_q(\alpha)$$

  Again, we know by definition 3.30 that $\rho_p(\alpha) = i$ and $\rho_q(\alpha) = j$. We distinguish three cases. First, $v(a) = i$. Then, $[\pi(\vec{l}, v, \nu)]_1(\textit{equiv}_{pq}(a)) = j$, since π swaps the values of a and $\textit{equiv}_{pq}(a)$, and it applies a data swap. Second, if $v(a) = j$, then we can conclude that $[\pi(\vec{l}, v, \nu)]_1(\textit{equiv}_{pq}(a)) = i$ by a similar argument. Third, if $v(a) \neq i, j$, then $[\pi(\vec{l}, v, \nu)]_1(\textit{equiv}_{pq}(a)) \neq i, j$, since the process swap interchanges the values of a and $\textit{equiv}_{pq}(a)$ and the data swap leaves these values unchanged.

- $g \equiv a = z$ or $g \equiv a \neq z$, where $a \in \textit{used}_\alpha$ and $z \in \mathbb{Z} \setminus \{0, 1, ..., s(\alpha) - 1\}$. If $a \in V^g$, then we can rewrite the equality above to:

  $$v(a) \sim z = \textit{eval}(a, A_q, [\pi(\vec{l}, v, \nu)]_1) \sim z$$

If $v(a) = i$ or $v(a) = j$, then the data swap part of the state swap changes the value of a. Since z is either larger than both i and j, or smaller than both i and j, the equality holds. If $v(a) \neq i, j$, then the value of a is left unchanged by the state swap. Therefore, the equality holds.

If $a \in V_p$, then we can rewrite the equality above to:

$$v(a) \sim z = \mathit{eval}(\mathit{equiv}_{pq}(a), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim z$$

If $v(a) = i$ or $v(a) = j$, then the process swap and data swap of the state swap act in such a way that the value of $\mathit{equiv}_{pq}(a)$ becomes j or i. Since z is either larger than both i and j, or smaller than both i and j, the equality holds. If $v(a) \neq i, j$, then the state swap acts in such a way that the value of $\mathit{equiv}_{pq}(a)$ becomes the value of a. Thus, the equality holds.

As for part (2) of the lemma, we only say that it can easily be proved by arguments very similar as above. The key observation is that the value of variable a is only possibly changed by the data swap, and *not* by the multiple process swap. ■

Lemma 3.39 *Consider a (β, n)-malformed integer guard $g \in IG(V^g \cup V_p, S_p)$, a state $(\vec{l}, v, \nu)$, a syntactical assignment swap π^s_{pq} and a state swap π. We can find a $n' \in \{0, ..., s(\beta) - 1\}$ such that for g', which results from replacing malformation n by n' in g, the following holds:*

(1) If π swaps A_p with A_q, then it is true that $\mathit{eval}(g', A_p, [(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}((\pi^s_{pq}(g'), A_q, [\pi(\vec{l}, v, \nu)]_1))$.

(2) If π does not swap A_p, then it is true that $\mathit{eval}(g', A_p, [(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}((g', A_p, [\pi(\vec{l}, v, \nu)]_1))$.

PROOF. A (β, n)-malformed integer guard can appear in two different shapes. First, it can appear as $a[i_0]...[n]...[i_m] \sim exp$, where the left side is malformed and the right side is well-formed. Second, it can appear as $a[i_0]...[n]...[i_m] \sim b[j_0]...[n]...[j_r]$, where both sides are malformed.

Let us consider the first case. With an argument as appears in the previous proof we can show that we must prove that we can find a n' such that

$$\begin{gathered}\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) \sim \mathit{eval}(exp, A_p, v) \\ = \\ \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim \\ \mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)\end{gathered}$$

By restrictions 2 and 3 on page 42 we know that exp is not equal to α and that it is not in used_α. Therefore, we can use lemma 3.33 to say that $\mathit{eval}(exp, A_p, v) =$

$\mathit{eval}(\pi^s_{pq}(exp), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Moreover, by lemma 3.34 we can find a n' such that

$$\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) = \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1)$$

This proves part (1) of the lemma for the first appearance of the malformed guard.

Let us consider the second case. Now we must prove that we can find a n' such that

$$\begin{gathered}\mathit{eval}(a[i_0]...[n']...[i_m], A_p, v) \sim \mathit{eval}(b[j_0]...[n']...[j_r], A_p, v)\\ =\\ \mathit{eval}(\pi^s_{pq}(a[i_0]...[n']...[i_m]), A_q, [\pi(\vec{l}, v, \nu)]_1) \sim\\ \mathit{eval}(\pi^s_{pq}(b[j_0]...[n']...[j_r]), A_q, [\pi(\vec{l}, v, \nu)]_1)\end{gathered}$$

Since both expressions are (β, n)-malformed, we can find one n' that proves this equivalence by lemma 3.34. This proves part (1) of the lemma.

Part (2) of the lemma.is very similar to the proof above, and we do not explicitly explain it. ■

Definition 3.40 (Syntactical clock assignment swap) *A syntactical swap of a clock assignment is a function $\pi^s_{pq} : CA(X^g \cup X_p) \to CA(X^g \cup X_q)$ defined as:*

$$\pi^s_{pq}(x := n) = \begin{cases} \mathit{equiv}_{pq}(x) := n & \text{if } x \in X_p \\ x := n & \text{otherwise} \end{cases}$$

Lemma 3.41 *Consider a clock reset $ca \in CA(X^g \cup X_p)$, two states $(\vec{l}, v, \nu)$ and $(\vec{l'}, v', \nu')$ such that $\nu' = \mathit{exec}((ca, \nu))$, a clock reset swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then $[\pi(\vec{l'}, v', \nu')]_2 = \mathit{exec}((\pi^s_{pq}(ca), [\pi(\vec{l}, v, \nu)]_2))$.

(2) If π does not swap A_p, then $[\pi(\vec{l'}, v', \nu')]_2 = \mathit{exec}((ca, [\pi(\vec{l}, v, \nu)]_2))$.

PROOF. According to definition 3.5, the clock reset is of the form $x := n$. We distinguish two cases for the proof of part (1) of the lemma:

- $x \in X^g$. We know by definition that $\nu'(y) = n$ if $y = x$ and $\nu(y)$ otherwise. Applying the state swap gives us that $[\pi(\vec{l'}, v', \nu')]_2(y) = n$ if $y = x$ and $[\pi(\vec{l}, v, \nu)]_2(y)$ otherwise, since $x \in X^g$. We now evaluate the right hand side of the equality. By definition: $\mathit{exec}(x := n, [\pi(\vec{l}, v, \nu)]_2)(y) = n$ if $y = x$, and $[\pi(\vec{l}, v, \nu)]_2(y)$ otherwise. We see that the definitions of both sides of the equality match.

- $x \in X_p$. First, we know by definition that $\nu'(y) = n$ if $y = x$ and $\nu(y)$ otherwise. Applying the state swap has as effect that the values of the local clocks of A_p and A_q are swapped. Therefore, we can expand the left hand side of the equality to:

$$[\pi(\vec{l'}, v', \nu')]_2(y) = \begin{cases} n & \text{if } y = \mathit{equiv}_{pq}(x), \text{ since } x \in X_p \\ [\pi(\vec{l}, v, \nu)]_2(y) & \text{otherwise} \end{cases}$$

It is straightforward to expand the right hand side of the equality to the same definition.

As for part (2) of the lemma we note that the state swap π can only change the values of local clocks by the process swaps. Since π does not swap A_p, the values of the local clocks of A_p remain unchanged. Therefore, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(x)$ for all $x \in X^g \cup X_p$, which proves part (2). ■

Definition 3.42 (Syntactical clock guard swap) *A syntactical swap of a clock guard is a function $\pi^s_{pq} : CG(X^g \cup X_p) \rightarrow CG(X^g \cup X_q)$ defined as:*

$$\pi^s_{pq}(cg) = \begin{cases} \pi_{pq}(x) \sim n & \text{if } cg \equiv x \sim n \\ \pi_{pq}(x) \sim \pi_{pq}(y) & \text{if } cg \equiv x \sim y \\ \pi_{pq}(x) \sim \pi_{pq}(y) + n & \text{if } cg \equiv x \sim y + n \end{cases}$$

where $x, y \in X^g \cup X_q$ and $n \in \mathbb{N}$, and the syntactical clock swap $\pi_{pq} : CG(X^g \cup X_p) \rightarrow CG(X^g \cup X_q)$ is defined as: $\pi_{pq}(x) = \mathit{equiv}_{pq}(x)$ if $x \in X_p$ and $\pi_{pq}(x) = x$ otherwise.

Lemma 3.43 *Consider a clock guard $cg \in CG(X^g \cup X_p)$, a state $(\vec{l}, v, \nu)$, a syntactical clock guard swap π^s_{pq} and a state swap π.*

(1) If π swaps A_p with A_q, then it is also true that $\mathit{eval}(cg, [(\vec{l}, v, \nu)]_2)$ equals $\mathit{eval}((\pi^s_{pq}(cg), [\pi(\vec{l}, v, \nu)]_2))$.

(2) If π does not swap A_p, then $\mathit{eval}(cg, [(\vec{l}, v, \nu)]_2) = \mathit{eval}(cg, [\pi(\vec{l}, v, \nu)]_2))$.

PROOF. We distinguish three cases for the proof of part (1) of the lemma:

- $cg \equiv x \sim n$. We prove that $\nu(x) \sim n$ equals $[\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim n$. We distinguish two cases. First, $x \in X^g$. Then, by definition the state swap π does not alter the value of x. Moreover, the syntactical clock swap does not change x. Thus, the right hand side can be written as $\nu(x) \sim n$. Second, $x \in X_p$. Then the state swap interchanges the values of the local clocks of process A_p and A_q. Thus, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(\mathit{equiv}_{pq}(x))$ and similarly, $[\pi(\vec{l}, v, \nu)]_2(\mathit{equiv}_{pq}(x)) = \nu(x)$. Using this, we rewrite the right hand side of the equality as follows:

$$\begin{aligned} [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim n &= [\pi(\vec{l}, v, \nu)]_2(\mathit{equiv}_{pq}(x)) \sim n \\ &= \nu(x) \sim n \end{aligned}$$

- $cg \equiv x \sim y$. To prove the equality, we must prove that $\nu(x) \sim \nu(y)$ equals

$$[\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x)) \sim [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$$

  In the previous item we have shown that $\nu(x) = [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$, if $x \in X^g \cup X_p$. This proves the equality.

- $cg \equiv x \sim y + n$. The proof of this case, again, is straightforward, since $\nu(x) = [\pi(\vec{l}, v, \nu)]_2(\pi_{pq}(x))$, if $x \in X^g \cup X_p$

As for part (2) of the lemma we note that the state swap π can only change the values of local clocks by the process swaps. Since π does not swap A_p, the values of the local clocks of A_p remain unchanged. Therefore, $[\pi(\vec{l}, v, \nu)]_2(x) = \nu(x)$ for all $x \in X^g \cup X_p$, which proves part (2). ■

Definition 3.44 (Syntactical synchronization swap) *A swap of a synchronization is a function* $\pi^s_{pq} : Sync(\Sigma, V^g \cup V_p, S_p) \rightarrow Sync(\Sigma, V^g \cup V_q, S_q)$ *defined as:*

$$\pi^s_{pq}(s) = \begin{cases} \tau & \text{if } s \equiv \tau \\ \sigma & \text{if } s \equiv \sigma \in \Sigma \\ \sigma[\pi^s_{pq}(i_0)]...[\pi^s_{pq}(i_m)] & \text{if } s \equiv \sigma[i_0]...[i_m] \end{cases}$$

where $\pi^s_{pq}(i_k)$ *is a syntactical integer expression swap of definition 3.32 which acts on the simple integer expression* i_k.

Lemma 3.45 *Assume that* π *is a state swap which swaps process* A_p *with process* A_q. *If* $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$, *then*

$$(src, \pi^s_{pq}(\sigma), (\pi^s_{pq}(ig), \pi^s_{pq}(cg)), (\pi^s_{pq}(ia), \pi^s_{pq}(ca)), dst) \in E_q$$

PROOF. The syntactical swaps, denoted by the overloaded function π^s_{pq}, convert local clocks, local variables and local constants of process A_p to equivalent local clocks, local variables and local constants of process A_q. These equivalence functions come forth from the template instantiation mechanism. One can see that this mechanism is such that the syntactical equivalence between the edges of the lemma holds. ■

In the next section we prove that the state swaps of definition 3.30 are automorphisms.

3.4.3 Proving Soundness

In this section we prove that the state swaps defined in definition 3.30 are automorphisms as defined in definition 3.1. We split the proof in five small lemmas. (We abbreviate π^α_{ij} to π, ξ^α_{ij} to ξ, and ζ^α_{ij} to ζ while maintaining the parameters α, i and j.)

Lemma 3.46 *If* π *is a state swap, then* s_0 *is the initial state iff* $\pi(s_0)$ *is the initial state.*

PROOF. We must prove that $\pi(s_0) = s_0$. First, we note that the initial locations of instances of the same template are the same. Since the process swaps only

swap instances of the same template and the data swaps leave the location vector untouched, we can conclude that the location vector remains the same.

Now let us consider the variable interpretation. First, we note that array entries are initialized to zero. Therefore, swapping equivalent arrays by a process swap, or swapping dimensions by a data swap does not have any effect. This leaves us the regular variables and we distinguish two cases.

- The regular variable $\in$ *used*$_\alpha$. If the variable is global, then only the data swap can change its value. However, it does not do that, since in restriction (2) (page 43) we required that the variable is initialized to a value not in $\{0, ..., s(\alpha) - 1\}$. If the variable is local to a template, then every instance of the template initializes the variable to the same value. This is due to the initialization requirement mentioned above.

- The regular variable $\notin$ *used*$_\alpha$. In this case, the value might only be changed due to a process swap if the variable is local. This does not change the variable interpretation, since the equivalent variable of the other template instance can only be initialized to the same value (otherwise, the variable would be $\in$ *used*$_\alpha$).

Therefore, state swap does not change the variable interpretation of the initial state.

Finally, consider the clock interpretation. Since all clocks are set to zero in the initial state, the clock interpretation does not change by swapping clock values. ■

Lemma 3.47 *(s, s') is a simple action transition iff $(\pi(s), \pi(s'))$ is a simple action transition.*

PROOF. We separately proof both sides of the equivalence, and we start with the implication to the right. Assume that $((\vec{l}, v, \nu), (\vec{l'}, v', \nu'))$ is a simple action transition as defined on page 44. We prove that $(\pi(\vec{l}, v, \nu), \pi(\vec{l'}, v', \nu'))$ is a simple action transition too. We split the proof in two parts.

- The transition is due to a well-formed edge $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$. We claim that if process A_p is not swapped by π, then this edge is still enabled. Otherwise, the edge

  $$(src, \pi^s_{pq}(\sigma), (\pi^s_{pq}(ig), \pi^s_{pq}(cg)), (\pi^s_{pq}(ia), \pi^s_{pq}(ca)), dst)$$

  which is an edge of A_q by lemma 3.45, is enabled. We prove all items of the definition:

  - From our main assumption we know that $l_p = src$ and $l'_p = dst$. If A_p is not swapped, then the location of A_p is not changed. Thus, obviously $[\pi(\vec{l}, v, \nu)]_{1_p} = src$ and $[\pi(\vec{l'}, v', \nu')]_{1_p} = dst$. If A_p is swapped with A_q, then the active locations of these processes are interchanged. Thus, $[\pi(\vec{l}, v, \nu)]_{1_q} = l_p = src$ and $[\pi(\vec{l'}, v', \nu')]_{1_q} = l'_p = dst$.

– Our main assumption is that $\mathit{eval}(ig, A_p, v) = \mathit{true}$ and $\mathit{eval}(cg, \nu) = \mathit{true}$. Assume that A_p is not swapped. By part (2) of lemma 3.38 we know that $\mathit{eval}(ig, A_p, [\pi(\vec{l}, v, \nu)]_1) = \mathit{eval}(ig, A_p, v) = \mathit{true}$. Similarly, by part (2) of lemma 3.43 we know that $\mathit{eval}(cg, [\pi(\vec{l}, v, \nu)]_2) = \mathit{eval}(cg, \nu) = \mathit{true}$.

  Now let us assume that A_p is swapped with A_q. We must prove that both $\pi^s_{pq}(ig)$ and $\pi^s_{pq}(cg)$ are true. By part (1) of lemma 3.38 we know that $\mathit{eval}(\pi^s_{pq}(ig), A_q, [\pi(\vec{l}, v, \nu)]_1)$ equals $\mathit{eval}(ig, A_p, v) = \mathit{true}$. Similarly, we know that $\mathit{eval}(\pi^s_{pq}(cg), [\pi(\vec{l}, v, \nu)]_2) = \mathit{eval}(cg, \nu) = \mathit{true}$ by part (1) of lemma 3.43.

  Concluding, in both cases the guards are satisfied.

– Our main assumption states that $v' = \mathit{exec}(ia, A_p, v)$ and also that $\nu' = \mathit{exec}(ca, \nu)$. Assume that A_p is not swapped. By part (2) of lemma 3.36 we know that $[\pi(\vec{l'}, v', \nu')]_1 = \mathit{exec}(ia, A_p, [\pi(\vec{l}, v, \nu)]_1)$. Similarly, by part (2) of lemma 3.41 we know that $[\pi(\vec{l'}, v', \nu')]_2 = \mathit{exec}(ca, [\pi(\vec{l}, v, \nu)]_2)$.

  Assume that A_p is swapped with A_q. By part (1) of lemma 3.36 we know that $[\pi(\vec{l'}, v', \nu')]_1 = \mathit{exec}(\pi^s_{pq}(ia), A_q, [\pi(\vec{l}, v, \nu)]_1)$. Similarly, we know that $[\pi(\vec{l'}, v', \nu')]_2 = \mathit{exec}(\pi^s_{pq}(ca), [\pi(\vec{l}, v, \nu)]_2)$ by part (1) of lemma 3.41.

  Concluding, in both cases the interpretations match the assignments.

– We know by our assumption that $\mathit{eval}(I_i(l_i), \nu) = \mathit{true}$ for all processes i. Now consider an invariant $I_p(l_p)$. If π does not swap A_p, then we know that $\mathit{eval}(I_p(l_p), [\pi(\vec{l}, v, \nu)]_2) = \mathit{eval}(I_p(l_p), [(\vec{l}, v, \nu)]_2) = \mathit{true}$ by lemma 3.43. Similarly, $\mathit{eval}(I_p(l_p), [\pi(\vec{l}, v, \nu')]_2) = \mathit{true}$.
  If π does swap A_p with A_q, then we know by lemma 3.43 that

  $$\mathit{eval}(\pi^s_{pq}(I_p(l_p)), [\pi(\vec{l}, v, \nu)]_2) = \mathit{eval}(I_p(l_p), [(\vec{l}, v, \nu)]_2) = \mathit{true}$$

  Similarly,

  $$\mathit{eval}(\pi^s_{qp}(I_q(l_q)), [\pi(\vec{l}, v, \nu)]_2) = \mathit{eval}(I_q(l_q), [(\vec{l}, v, \nu)]_2) = \mathit{true}$$

  The syntactical swap converts the invariants: $\pi^s_{pq}(I_p(l_p)) = I_q(l_q)$, and vice versa. Thus, $\mathit{eval}(I_p(l_p), [\pi(\vec{l}, v, \nu)]_2) = \mathit{true}$ and the same holds for $I_q(l_q)$. We can repeat the same argument for $(\vec{l'}, v', \nu')$.

  Concluding, all invariants are still satisfied after the state swap.

– Observe that the state swap *permutes* the location vector. Therefore, the count of committed locations does not change. Moreover, if A_p is swapped with A_q, then the active locations are interchanged. Thus, l_p is active iff $[\pi(\vec{l}, v, \nu)]_{1_q}$ is active. (The processes originate from the same template, thus the same locations are committed.)

- The transition is due to an (β, n)-malformed edge $(src, \sigma, (ig, cg), (ia, ca), dst) \in E_p$. We claim that if A_p is not swapped by π, then we can use this edge, or we can find another edge in E_p which proves that $(\pi(s), \pi(s'))$ is a simple action transition. The proof follows the same structure as in the previous item, except we use the lemma's 3.37 and 3.39 in conjunction with restriction (3) on page 43. On the other hand, if A_p is swapped with A_q by π, then we can find an edge in E_q which proves or claim. Again, we need the lemma's 3.37 and 3.39 in conjunction with restriction (3).

The implication to the left can easily be proved. Assume that $(\pi(s), \pi(s'))$ is a simple action transition. Above we have proved that if (s, s') is a simple action transition, then $(\pi(s), \pi(s'))$ is a simple action transition. Thus, we can say that $(\pi \circ \pi(s), \pi \circ \pi(s'))$ is a simple action transition. In lemma 3.31 we have proved that $\pi \circ \pi(s) = s$. Therefore, (s, s') is a simple action transition. ■

Lemma 3.48 *(s, s') is a σ action transition iff $(\pi(s), \pi(s'))$ is a σ action transition.*

PROOF. The proof of this lemma is very similar to the prove given for lemma 3.47. Therefore, we do not explicitly explain it. ■

Lemma 3.49 *(s, s') is a δ delay transition iff $(\pi(s), \pi(s'))$ is a δ delay transition.*

PROOF. We defined a δ delay transition on page 45. As in the previous proofs, we first prove the implication to the right. The implication to the left follows without effort.

Assume that $((\vec{l}, v, \nu), (\vec{l}, v, \nu'))$ is a δ delay transition. We consider every item of the definition separately:

- We know by our assumption that $\nu'(x) = \nu(x) + \delta$ for all clocks x. Now consider a clock y and assume that it is not swapped. Thus: $[\pi(\vec{l}, v, \nu)]_2(y) = \nu(y)$, and $[\pi(\vec{l}, v, \nu')]_2(y) = \nu'(y)$. Since we assumed that $\nu'(x) = \nu(x) + \delta$ for all clocks x, we can conclude that $[\pi(\vec{l}, v, \nu')]_2(y) = [\pi(\vec{l}, v, \nu)]_2(y) + \delta$.

  If y is swapped, then $[\pi(\vec{l}, v, \nu)]_2(y) = \nu(z)$ and $[\pi(\vec{l}, v, \nu')]_2(y) = \nu'(z)$ for some equivalent clock z. Since we assumed that $\nu'(x) = \nu(x) + \delta$ for all clocks x, we know that $[\pi(\vec{l}, v, \nu')]_2(y) = [\pi(\vec{l}, v, \nu)]_2(y) + \delta$.

- See the fourth item of the first bullet in the proof of lemma 3.47.

- The state swap only *permutes* the active locations. Therefore, no committed locations become active by swapping.

- By our main assumption we know that if there exists an i such that $lt_i(l_i) =$ *urgent*, then no state r exists such that (s, r) is a simple or σ action transition.

We must prove that if there exists an i such that $lt_i(l_i) = \textit{urgent}$, then no state r' exists such that $(\pi(s), r')$ is a simple or σ action transition. It suffices to prove the right hand side of the implication. Therefore, assume that this state r' does exist. From lemma 3.47 and lemma 3.48 we know that $(\pi \circ \pi(s), \pi(r'))$ also is a simple or σ action transition. By lemma 3.31 we can conclude that $(s, \pi(r'))$ is a simple or σ action transition. Thus, the state r mentioned above does exist, namely $t = \pi(r')$. From this contradiction we can conclude that the state r' does not exist.

- The argument is similar to the one in the previous item.

■

The following corollary is the main result of this paper.

Corollary 3.50 (Soundness) *Every state swap is an automorphism.*

PROOF. By combination of lemmas 3.31, 3.46, 3.47, 3.48 and 3.49. ■

We can use these automorphisms to construct a normal form operator θ, which can be used by the forward exploration algorithm, see section 3.2. In general, computing a canonical representative of a symmetry class (the so-called orbit problem) is at least as difficult as the graph isomorphism problem, for which currently no polynomial time algorithms exist [38]. Therefore, it is practically more useful to use a fast, but non-canonical, normal form operator. This increases the memory usage in comparison with a canonical normal form operator, but is, very probably, much faster. There are many possibilities for a non-canonical normal form operator (for example, see [26]). These are all based on minimizing the state using the automorphisms. The challenge is to find a computationally efficient normal form operator, which improves on the non-symmetric tool in most situations. Since we find reasoning about the gain of normal form operators for different models very difficult, we like to address this with experimental research which can be conducted after the prototype implementation.

As a final note we mention that the so-called state properties, which are used to define sets of states, should also be taken into account by our normal form operator. We do not expect this to be a difficult problem.

3.5 Conclusions

We have proposed an enhancement of the model checker UPPAAL, which exploits structural symmetries to reduce the searchable state space. In Section 3.2 we summarized work of Ip and Dill [69] which we used for our symmetry reduction technique. In Section 3.3 we have proposed a syntactical adjustment of the system description language of UPPAAL version 3.2. More precisely, we have added the well-known scalarset data type, and multi-dimensional arrays of integer variables

and channels. In Section 3.4 we used these scalarsets to extract automorphisms on the state graph of our models. The main result of this paper is corollary 3.50, which states that our technique is sound.

Clearly, the soundness proof given in this chapter heavily depends on the syntax of the modeling language. Therefore, even the smallest change to the language in principle invalidates the proof. This poses a problem since the syntax of modeling languages often is changing and growing rather quickly. (This certainly is the case for UPPAAL: version 3.2 differs greatly from the current version 3.4 and will differ even more from the next major release.) There are at least two solutions to this problem. First, one can formalize the current syntax and soundness proof in a theorem prover such as PVS [89]. Probably, the construction of the formalized proof is not very hard since the manual proof is not "deep" but does consist of many case-distinctions. When small changes to the syntax are made, PVS might be able to rerun the existing proof automatically. Second, one might be able to think of certain meta-theorems that ensure the soundness of a whole class of modeling languages. Another issue concerning the correctness of the tool is that often several techniques to improve the scalability are applied simultaneously. It must therefore be ascertained that these techniques do not influence each other in an unsound way.

Future work includes the implementation of the proposed technique. As experiences of Ip and Dill already showed, the actual implementation of the computation of the representative of a symmetry class is very important [69]. Canonical representatives minimize the space usage, but can be very costly in time. The feasibility of non-canonical representatives, however, might vary per model. Therefore, experiments with different algorithms to compute non-canonical representatives and a fairly large set of different models are necessary to assess the effectiveness of scalarsets in a dense-time setting.

Acknowledgements. The author thanks Frits Vaandrager for commenting on earlier versions of this chapter.

Chapter 4

Adding Symmetry Reduction to Uppaal

MARTIJN HENDRIKS

GERD BEHRMANN

KIM LARSEN

PETER NIEBERT

FRITS VAANDRAGER

Abstract. We describe a prototype extension of the UPPAAL real-time model checking tool with symmetry reduction. The symmetric data type *scalarset*, which is also used in the MURφ model checker, was added to UPPAAL's system description language to support the easy static detection of symmetries. Our prototype tool uses *state swaps*, described and proved sound earlier by Hendriks, to reduce the space and memory consumption of UPPAAL. Moreover, under certain assumptions the reduction strategy is *canonical*, which means that the symmetries are optimally used. For all examples that we experimented with (both academic toy examples and industrial cases), we obtained a drastic reduction of both computation time and memory usage, exponential in the size of the scalar sets used.

4.1 Introduction

The state space explosion problem often renders the mechanical verification of realistic systems practically impossible: there just is not enough time or memory available. As a consequence, much research has been directed at finding techniques to fight the state space explosion. One such a technique is the exploitation of behavioral symmetries [66, 98, 71, 68, 48, 38]. The exploitation of *full* symmetries can be particularly profitable, since its gain can approach a factorial magnitude.

There are many timed systems which clearly exhibit full symmetry, e.g., Fischer's mutual exclusion protocol [1], the CSMA/CD protocol [99, 107], industrial audio/video protocols [53], and distributed algorithms, for instance [10]. Motivated by these examples, the work presented in [55] describes how UPPAAL, a model checker for networks of timed automata [16, 7, 6], can be enhanced with symmetry reduction. The present paper puts this work to practice: a prototype of UPPAAL with symmetry reduction has been implemented[1]. The symmetric data type *scalarset*, which was introduced in the MURφ model checker [45], was added to UPPAAL's system description language to support the easy static detection of

This chapter is a literal copy of [58], which on its turn is the extended version of [57].

[1] We currently are working on the implementation of symmetry reduction in the UPPAAL main line. We expect that symmetry reduction comes publicly available from version 3.5.0.

symmetries. Furthermore, the *state swaps* described and proved sound in [55] are used to reduce the space and time consumption of the model checking algorithm. The reduction strategy is optimal under certain assumptions that essentially concern the discrete part of the state only. Thus, the dense time domain does not add extra complexity to the symmetry reduction technique. Run-time data is reported for the examples mentioned above, showing that symmetry reduction in a timed setting can be very effective.

Related work. Symmetry reduction is a well-known technique to reduce the resource requirements for model checking algorithms, and it has been successfully implemented in model checkers such as MURφ [45, 68], SMV [83], and SPIN [65, 26]. As far as we know, the only model checker for timed systems that exploits symmetry is RED [102, 104]. The symmetry reduction technique used in RED, however, gives an over approximation of the reachable state space (this is called the *anomaly of image false reachability* by the authors). Therefore, RED can only be used to ensure that a state is *not* reachable when it is run with symmetry reduction, whereas symmetry enhanced UPPAAL can be used to ensure that a state is reachable, or that it is not reachable.

Contribution. We have added symmetry reduction as used within MURφ, a well-established technique to combat the state space explosion problem, to the real-time model checking tool UPPAAL. For researchers familiar with model checking it will come as no surprise that this combination can be made and indeed leads to a significant gain in performance. Still, the effort required to actually add symmetry reduction to UPPAAL turned out to be substantial. The soundness of the symmetry reduction technique that we previously developed for UPPAAL does not follow trivially from the work of Ip and Dill [68] since the description languages of UPPAAL and MURφ, from which symmetries are extracted automatically, are quite different. In fact, the proof that symmetry reduction for UPPAAL is sound takes up more than 20 pages in [55]. The main technical contribution of the present work is an efficient algorithm for the computation of a representative that – under certain assumptions – is optimal. This is not trivial due to UPPAAL's symbolic representation of sets of clock valuations. Many timed systems exhibit symmetries that can be exploited by our methods. For all examples that we experimented with, we obtained a drastic reduction of both computation time and memory usage, exponential in the size of the scalar sets used.

Outline. Section 4.2 presents a very brief summary of model checking and symmetry reduction in general, while Sections 4.3 and 4.4 introduce symmetry reduction for the UPPAAL model checker in particular. In Section 4.5, we present run-time data of UPPAAL's performance with and without symmetry reduction, and Section 4.6 summarizes and draws conclusions.

4.2 Model Checking and Symmetry Reduction

This section briefly summarizes the theory of symmetry presented in [68], which we reuse in a timed setting since (i) it has proved to be quite successful, and (ii) it is designed for reachability analysis, which is the main purpose of the UPPAAL model checker. We simplify (and in fact generalize) the presentation of [68] using the concept of bisimulations.

In general, a transition system is a tuple (Q, Q_0, Δ), where Q is a set of states, $Q_0 \subseteq Q$ is a set of initial states, and $\Delta \in Q \times Q$ is a transition relation between states. Figure 4.1 depicts a general forward reachability algorithm which, under the assumption that Q is finite, computes whether there exists a reachable state q that satisfies some given property ϕ (denoted by $q \models \phi$).

```
(1)   passed := ∅
(2)   waiting := Q0
(3)   while waiting ≠ ∅ do
(4)         get q from waiting
(5)         if q ⊨ φ then return YES
(6)         else if q ∉ passed then
(7)               add q to passed
(8)               waiting := waiting ∪ { q' ∈ Q | (q, q') ∈ Δ }
(9)         fi
(10)  od
(11)  return NO
```

Figure 4.1: A general forward reachability analysis algorithm.

Due to the state space explosion problem, the number of states of a transition system frequently gets too big for the above algorithm to be practical. We would like to exploit structural properties of transition systems (in particular symmetries) to improve its performance. Here the well-known notion of bisimulation comes in naturally:

Definition 4.1 (Bisimulation) *A* bisimulation *on a transition system* (Q, Q_0, Δ) *is a relation* $R \subseteq Q \times Q$ *such that for all* $(q, q') \in R$*,*

1. $q \in Q_0$ *if and only if* $q' \in Q_0$*,*
2. *if* $(q, r) \in \Delta$ *then there is an* r' *such that* $(q', r') \in \Delta$ *and* $(r, r') \in R$*,*
3. *if* $(q', r') \in \Delta$ *then there exists an* r *such that* $(q, r) \in \Delta$ *and* $(r, r') \in R$*.*

Suppose that, before starting the reachability analysis of a transition system, we know that a certain equivalence relation $\approx$ is a bisimulation and respects the predicate ϕ in the sense that either all states in an equivalence class satisfy ϕ or none of them do. Then, when doing reachability analysis, it suffices to store and explore only a single element of each equivalence class. To implement the state

space exploration, a *representative function* θ may be used that converts a state to a representative of the equivalence class of that state:

$$\forall_{q \in Q}\ (q \approx \theta(q)) \tag{4.1}$$

Using θ, we may improve the algorithm in Figure 4.1 by replacing lines 2 and 8, respectively, by:

(2) *waiting* $:= \{\, \theta(q) \mid q \in Q_0 \,\}$
(8) *waiting* $:=$ *waiting* $\cup \{\, \theta(q') \mid (q, q') \in \Delta \,\}$

It can easily be shown that the adjusted algorithm remains correct: for all (finite) transition systems the outcomes of the original and the adjusted algorithm are equal. If the representative function is "good", which means that many equivalent states are projected onto the same representative, then the number of states to explore, and consequently the size of the *passed* set, may decrease dramatically. However, in order to apply the approach, the following two problems need to be solved:

- A suitable bisimulation equivalence $\approx$ that respects ϕ needs to be statically derived from the system description.
- An appropriate representative function θ needs to be constructed that satisfies equation (4.1), and that can be computed efficiently. Ideally, θ satisfies $q \approx q' \Rightarrow \theta(q) = \theta(q')$, in which case it is called *canonical*.

In this paper, we use symmetries to solve these problems. As in [68], the notion of *automorphism* is used to characterize symmetry within a transition system. This is a bijection on the set of states that (viewed as a relation) is a bisimulation. Phrased alternatively:

Definition 4.2 (Automorphism) *An* automorphism *on some transition system, say* (Q, Q_0, Δ)*, is a bijection* $h : Q \to Q$ *such that*

1. $q \in Q_0$ *if and only if* $h(q) \in Q_0$ *for all* $q \in Q$*, and*
2. $(q, q') \in \Delta$ *if and only if* $(h(q), h(q')) \in \Delta$ *for all* $q, q' \in Q$*.*

Let H be a set of automorphisms, let $\mathbf{id}$ be the identity function on states, and let $G(H)$ be the closure of $H \cup \{\mathbf{id}\}$ under inverse and composition. It can be shown that $G(H)$ is a group, and it induces a bisimulation equivalence relation $\approx$ on the set of states as follows:

$$q \approx q' \iff \exists_{h \in G(H)}\ (h(q) = q') \tag{4.2}$$

We introduce a symmetric data type to let the user explicitly point out the symmetries in the model. Simple static checks can ensure that the symmetry that is

pointed out is not broken. Our approach to the second problem of coming up with good representative functions consists of "sorting the state" w.r.t. some ordering relation on states using the automorphisms. For instance, given a state q and a set of automorphisms, find the smallest state q' that can be obtained by repeatedly applying automorphisms and their inverses to q. It is clear that such a θ satisfies Equation 4.1, since it is constructed from the automorphisms only.

4.3 Adding Scalarsets to Uppaal

The tool UPPAAL is a model checker for networks of timed automata extended with discrete variables (bounded integers, arrays) and blocking, binary synchronization as well as non-blocking broadcast communication (see for instance [16]). In the remainder of this section we illustrate by an example UPPAAL's description language extended with a *scalarset* type constructor allowing symmetric data types to be syntactically defined. Our extension is based on the notion of scalarset first introduced by Ip and Dill in the finite-state model checking tool MURφ [45, 68]. Also our extension is based on the C-like syntax to be introduced in the forthcoming version 4.0 of UPPAAL.

To illustrate our symmetry extension of UPPAAL we consider Fischer's mutual exclusion protocol. This protocol consists of n processes, identical up to their unique process identifiers. The purpose of the protocol is to ensure mutual exclusion on the critical sections of the processes. This is accomplished by letting each process write its identifier (`pid`) in a global variable (`id`) before entering its critical section. If after some given lower time bound (say 2) `id` still contains the `pid` of the process, then it may enter its critical section.

A scalarset of size n may be considered as the subrange $\{0, 1, \ldots, n-1\}$ of the natural numbers. Thus, the n process identifiers in the protocol can be modeled using a scalarset with size n. In addition to the global variable `id`, we use the array `active` to keep track of all active locations of the processes[2]. Global declarations are the following:

```
typedef scalarset[3] proc_id;  // a scalarset type with size 3
proc_id id;                    // declaration of a proc_id
                               //   variable
bool set;                      // declaration of a boolean
int active[proc_id];           // declaration of an array
                               //   indexed by proc_id
```

The first line defines `proc_id` to be a scalarset type of size 3, and the second line declares `id` to be a variable over this type. Thus `scalarset` is viewed as a type constructor. In the last line we show a declaration of an array indexed by elements of the scalarset `proc_id`.

[2]This array is actually redundant and not present in the standard formulations of the protocol. It is, however, useful for showing important aspects of our extension.

At this point the only thing missing is the declaration of the actual processes in the system. In the description language of UPPAAL, processes are obtained as instances of parameterized process templates. In general, templates may contain several different parameters (e.g. bounded integers, clocks, and channels). In our extension we allow in addition the use of scalarsets as parameters. In the case of Fischer's protocol the processes of the system are given as instances of the template depicted in Figure 4.2.

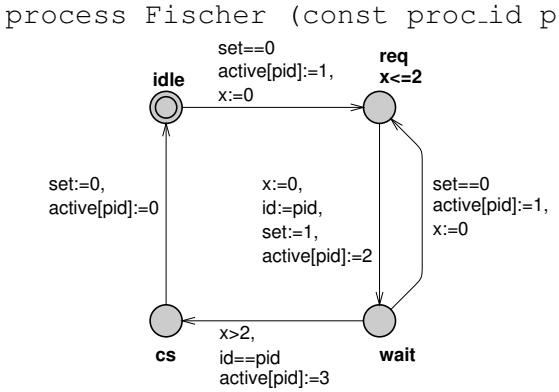

Figure 4.2: The template for Fischer's protocol.

The template has one local clock, `x`, and no local variables. Note that the header of the template defines a (constant) scalarset parameter `pid` of type `proc_id`. Access to the critical section `cs` is governed by suitable updates and tests of the global scalarset variable `id` together with upper and lower bound time constraints on when to proceed from requesting access (`req`) respectively proceed from waiting for access (`wait`). Note that all transitions update the array `active` to reflect the current active location of the process. The instantiation of this template and declaration of all three process in the system can be done as follows:

```
FischerProcs = forall i in proc_id : Fischer(i);
system FischerProcs;
```

The `forall` construct iterates over all elements of a declared scalarset type. In this case the iteration is over `proc_id` and a set of instances of the template `Fischer` is constructed and bound to `FischerProcs`. In the second line the final system is defined to be precisely this set.

4.4 Using Scalarsets for Symmetry Reduction

This section first presents a method to extract automorphisms from a UPPAAL system description using the new scalarset type. These automorphisms can be used for computation of the representative of a state as described in Section 4.2. Second, a total preorder is introduced on the individual clocks of zones that are generated

during the exploration of the state space. Third, a representative function is defined that uses this preorder on clocks. The main technical result is a proof that this function is canonical under certain assumptions. The representative function may not be canonical without all these assumptions, but it certainly is sound.

4.4.1 Extraction of Automorphisms

The extended syntax as described in the previous section enables us to derive the following information from a system description:

1. A set Ω of scalarset types.
2. For each $\alpha \in \Omega$: (i) a set V_α of regular variables of type α[3], and (ii) a set D_α of pairs (a, n), where a is an integer variable or clock array and n is a dimension of a that is to be indexed by α elements such that $\alpha \neq \beta \Rightarrow D_\alpha \cap D_\beta = \emptyset$.
3. A partial mapping $\gamma : P \times \Omega \hookrightarrow \mathbb{N}$ that gives for each process p and scalarset α the element of α with which p is instantiated. This mapping is defined by quantification over scalarsets in the process definition section.

A UPPAAL state is a tuple $(\vec{l}, v, Z)$, where $\vec{l}$ is the location vector, v is the integer variable valuation, and Z is a *zone*. A zone is a set of *clock valuations*, i.e., functions $\nu : X \rightarrow \mathbb{R}_+$ where X is the set of clocks and $\mathbb{R}_+$ denotes the set of non-negative real numbers. Zones are represented in UPPAAL by *difference bounded matrices* (DBMs) [18, 43]. Concretely, the location vector and variable valuation are implemented by arrays of integers, and the DBM is implemented by a matrix of integers. The UPPAAL state representation assumes that every process has a fixed index in the location vector, every regular integer variable and every entry of an integer array variable has a fixed index in the variable valuation, and that every clock has a fixed index in the DBM. Thus, there are three injections, all denoted by ρ, that map processes, integer variables, and clocks to indices.

Next, we introduce the notion of *substate* for every scalarset element. Informally, the substate of element i of the scalarset α is a triple containing (i) indices of processes that have been instantiated with element i of α, (ii) indices of variables (or array entries) that are associated with element i of α, and (iii) indices of clocks that are associated with element i of α. These substates can statically be derived, and do not change during the state space exploration.

Definition 4.3 (Substate) *Let i be an element of the scalarset α. The substate of this element, denoted by substate(α, i), is a tuple $(\vec{l}, \vec{v}, \vec{c})$, where*

[3]The soundness proof in [55] uses the so-called *used*$_\alpha$ set for this set. Moreover, it is assumed that arrays of integer variables cannot be part of this *used*$_\alpha$ set, which is needed for the soundness proof of the state swaps. Since this soundness proof is reused in the present paper, the assumption that V_α only consists of regular variables should be made. This assumption, however, can probably be dropped.

- $\vec{l}$ *is the* ordered *sequence of indices of all processes in* $\{p \mid \gamma(p, \alpha) = i\}$.
- $\vec{v}$ *is the* ordered *sequence of indices of*
  - *the local integer variables of all processes* p *that satisfy* $\gamma(p, \alpha) = i$.
  - *integer array entries that are associated with the* i*-th element of* α*, i.e., the entry* $b[j_1] \cdots [j_k] \cdots [j_n]$ *is associated with the* i*-th element of* α *if and only if* $(b, k) \in D_\alpha$ *and* $j_k = i$.
- $\vec{c}$ *is the* ordered *sequence of indices of*
  - *local clocks of all processes* p *that satisfy* $\gamma(p, \alpha) = i$.
  - *clock array entries that are associated with the* i*-th element of* α*, i.e., the entry* $c[j_1] \cdots [j_k] \cdots [j_n]$ *is associated with the* i*-th element of* α *if and only if* $(c, k) \in D_\alpha$ *and* $j_k = i$.

We use these substates to define the automorphisms, and to this end we need the next assumptions. We note that the definition of automorphisms that appears in [55] does not need the assumptions below and therefore is more general, but also more complicated. Moreover, these assumptions are also needed in the proof that the representative function is canonical.

Assumption 4.4 (Basic assumptions)

(1) Local arrays are not *indexed by scalarsets and* V_α *only contains* global *variables.*

(2) At most one dimension of an array can be indexed by a scalarset, i.e., $|\{(b, n) \in D_\alpha \mid \alpha \in \Omega \wedge n \in \mathbb{N}\}| \leq 1$ *for all arrays of integer and clock variables* b.

(3) A process can be associated with at most one scalarset element, i.e., $|\{(\alpha, i) \mid \gamma(p, \alpha) = i \wedge \alpha \in \Omega\}| \leq 1$ *for all processes* p.

The contributions to the state of different substates are completely disjoint, which is formalized by the following lemma.

Lemma 4.5 *If* $\alpha \neq \beta$ *or* $i \neq j$*, then substate*(α, i) *and substate*(β, j) *are disjoint.*

PROOF. Let *substate*$(\alpha, i) = (\vec{l}_1, \vec{v}_1, \vec{c}_1)$ and let *substate*$(\beta, j) = (\vec{l}_2, \vec{v}_2, \vec{c}_2)$.

1. $\vec{l}_1$ and $\vec{l}_2$ share no indices. For the proof, assume that they do, i.e., by Definition 4.3 a process p exists such that $\gamma(p, \alpha) = i$ and $\gamma(p, \beta) = j$. We see, however, that p now is associated with two scalarset elements, since $\alpha \neq \beta$ or $i \neq j$, which contradicts the third item of Assumption 4.4. Therefore, $\vec{l}_1$ and $\vec{l}_2$ share no indices.

2. $\vec{v}_1$ and $\vec{v}_2$ share no indices. In the previous item we proved that the substates refer to disjoint sets of processes. Thus, the sets of local variables of these sets of processes also are disjoint. Now consider an array entry $b[h_1]\dots[h_n]$ and assume that it is associated with the i-th element of α and with the j-th element of β. This means (see Definition 4.3) that some k exists such that $(b,k) \in D_\alpha \wedge h_k = i$ and some k' exists such that $(b,k') \in D_\beta \wedge h_{k'} = j$. From our assumption that $\alpha \neq \beta \vee i \neq j$ we can easily prove that $k \neq k'$: (1) if $\alpha \neq \beta$, then $D_\alpha \cap D_\beta = \emptyset$. Therefore, $(b,k) \neq (b,k')$ and clearly $k \neq k'$. (2) if $i \neq j$, then $h_k \neq h_{k'}$ and clearly $k \neq k'$. Thus, two dimensions of b are indexed by scalarsets which clearly contradicts the second item in Assumption 4.4. Therefore, $\vec{v}_1$ and $\vec{v}_2$ share no indices.

3. $\vec{c}_1$ and $\vec{c}_2$ share no indices. We can prove this by a similar argument as in the previous case.

■

Consider some substate $(\vec{l}_1, \vec{v}_1, \vec{c}_1)$ and a state $q = (\vec{l}, v, Z)$. We can *project* the state to this substate: $[\![\vec{l}_1]\!]_q$ is obtained from $\vec{l}_1$ by replacing every index by the encoding of the associated active location (according to q of course). The projection of q to $\vec{v}_1$, denoted by $[\![\vec{v}_1]\!]_q$, is obtained similarly. If we use the notation $[\vec{l}]_k$ to refer to the k-th element in the sequence $\vec{l}$, then for all k:

$$[\![\vec{l}_1]\!]_q = \vec{l}'_1 \quad \text{where } [\vec{l}'_1]_k = [\vec{l}]_{[\vec{l}_1]_k} \tag{4.3}$$

$$[\![\vec{v}_1]\!]_q = \vec{v}'_1 \quad \text{where } [\vec{v}'_1]_k = [v]_{[\vec{v}_1]_k} \tag{4.4}$$

Note that the location and variable projections can be ordered using the lexicographical order on sequences of numbers. We cannot easily define the projection of a state to the clock indices, since the state contains a *set* of clock valuations. For a single clock valuation ν, however, we define the projection of ν to a clock index vector $\vec{c}_1$ as follows:

$$[\![\vec{c}_1]\!]_\nu = \vec{c}'_1 \quad \text{where } [\vec{c}'_1]_k = \nu(\rho^{-1}([\vec{c}_1]_k)) \tag{4.5}$$

Clock valuation projections can be ordered using the lexicographical order on sequences of numbers. In the next subsection a preorder on clocks is defined that enables us to compare the projections of the clock parts of substates for any given state.

The next assumption formalizes the correspondence between substates of different elements of a scalarset. This correspondence is ensured by the implementation of UPPAAL, and is needed to define the state swaps that are used for the computation of representatives.

Assumption 4.6 (Detailed assumptions) *Let substate*$(\alpha, i) = (\vec{l}_1, \vec{v}_1, \vec{c}_1)$*, and let substate*$(\alpha, j) = (\vec{l}_2, \vec{v}_2, \vec{c}_2)$.

1. *The length of $\vec{l}_1$ equals the length of $\vec{l}_2$, and $i < j \Leftrightarrow [\vec{l}_1]_k < [\vec{l}_2]_k$.*

2. *The length of $\vec{v}_1$ equals the length of $\vec{v}_2$, and $i < j \Leftrightarrow [\vec{v}_1]_k < [\vec{v}_2]_k$, and $[\vec{v}_1]_k$ and $[\vec{v}_2]_k$ refer to equivalent variables:*

   - *$[\vec{v}_1]_k$ is the index of the local variable b of the process $\rho^{-1}([\vec{l}_1]_q)$ if and only if $[\vec{v}_2]_k$ is the index of the local variable b of the process $\rho^{-1}([\vec{l}_2]_q)$.*
   - *$[\vec{v}_1]_k$ is the index of the array entry $b[h_1]\dots[h_{p-1}][i][h_{p+1}]\dots[h_q]$ if and only if $[\vec{v}_2]_k$ is the index of $b[h_1]\dots[h_{p-1}][j][h_{p+1}]\dots[h_q]$.*

3. *The length of $\vec{c}_1$ equals the length of $\vec{c}_2$, and $i < j \Leftrightarrow [\vec{c}_1]_k < [\vec{c}_2]_k$, and $[\vec{c}_1]_k$ and $[\vec{c}_2]_k$ refer to equivalent clocks (defined as above).*

Assumptions 4.4 and 4.6 enable us to define so-called *state swaps*[4].

Definition 4.7 (State swap) *Consider two distinct elements, say i and j, of some scalarset α. Let substate$(\alpha, i) = (\vec{l}_1, \vec{v}_1, \vec{c}_1)$, and let substate$(\alpha, j) = (\vec{l}_2, \vec{v}_2, \vec{c}_2)$. The* state swap *is defined as swap$^{\alpha}_{i,j}((\vec{l}, v, Z)) = (\vec{l'}, v', Z')$, where:*

- *$\vec{l'}$ is defined as follows for all k:*

$$[\vec{l'}]_k = \begin{cases} [\vec{l}]_k & \text{if } k \notin \vec{l}_1 \text{ and } k \notin \vec{l}_2 \\ [\vec{l}]_m & \text{if } k = [\vec{l}_1]_n, \text{ where } m = [\vec{l}_2]_n \\ [\vec{l}]_m & \text{if } k = [\vec{l}_2]_n, \text{ where } m = [\vec{l}_1]_n \end{cases}$$

- *v' can be defined as follows (remember that V_α only contains global non-array variables):*

$$[v']_k = \begin{cases} [v]_k & \text{if } k \notin \vec{v}_1 \text{ and } k \notin \vec{v}_2 \text{ and } \rho^{-1}(k) \notin V_\alpha \\ i & \text{if } \rho^{-1}(k) \in V_\alpha \text{ and } [v]_k = j \\ j & \text{if } \rho^{-1}(k) \in V_\alpha \text{ and } [v]_k = i \\ [v]_m & \text{if } k = [\vec{v}_1]_n, \text{ where } m = [\vec{v}_2]_n \\ [v]_m & \text{if } k = [\vec{v}_2]_n, \text{ where } m = [\vec{v}_1]_n \end{cases}$$

- *$Z' = \{s(\nu) \mid \nu \in Z\}$, where the clock value swap s is defined for all clocks x as:*

$$(s(\nu))(x) = \begin{cases} \nu(x) & \text{if } \rho(x) \notin \{[\vec{c}_1]_k, [\vec{c}_2]_k\} \text{ for all } k \\ \nu(\rho^{-1}([\vec{c}_1]_k)) & \text{if } \rho(x) = [\vec{c}_2]_k \\ \nu(\rho^{-1}([\vec{c}_2]_k)) & \text{if } \rho(x) = [\vec{c}_1]_k \end{cases}$$

[4] A more general definition that covers the current definition, but does not need Assumption 4.4 appears in [55]. E.g., that definition also covers the case when an array is indexed by multiple scalarsets.

Note that by definition $\mathit{swap}^{\alpha}_{i,j}(q) = \mathit{swap}^{\alpha}_{j,i}(q)$. A number of syntactic checks have been identified in [55] that ensure that the symmetry as suggested by the scalarsets is not broken. These checks are very similar to those originally identified for the MURφ verification system [68]. For instance, it is not allowed to use variables of a scalarset type for arithmetical operations such as addition. The next soundness theorem has been proved in [55] (provided that the symmetry is not broken)[5].

Theorem 4.8 (Soundness) *Every state swap is an automorphism.*

As a result, the representative function θ can be implemented by minimization of the state using the state swaps. Note that every state swap resembles a transposition of two scalarset elements. Hence, the equivalence classes induced by the state swaps originating from a scalarset with size n consist of at most $n!$ states. The maximal theoretical gain that can be achieved using the state swaps therefore is in the order of a factor $n!$.

Consider the instance of Fischer's mutual exclusion protocol as described in the previous section with three processes. There are three state swap functions: $\mathit{swap}^{\mathtt{proc\_id}}_{0,1}$, $\mathit{swap}^{\mathtt{proc\_id}}_{0,2}$ and $\mathit{swap}^{\mathtt{proc\_id}}_{1,2}$. Now consider the following state of the model (the active location of the i-th process is given by $[\vec{l}]_i$ and the local clock of this process is given by x_i; also note that the zone Z only contains one clock valuation):

$$\begin{array}{lcl} \vec{l} & : & (\mathtt{idle}, \mathtt{wait}, \mathtt{cs}) \\ v & : & \mathtt{id} = 2,\ \mathtt{set} = 1,\ \mathtt{active}[0] = 0,\ \mathtt{active}[1] = 2,\ \mathtt{active}[2] = 3 \\ Z & : & x_0 = 4,\ x_1 = 3,\ x_2 = 2.5 \end{array}$$

When we apply $\mathit{swap}^{\mathtt{proc\_id}}_{0,2}$ to this state, the result is the following state:

$$\begin{array}{lcl} \vec{l} & : & (\mathtt{cs}, \mathtt{wait}, \mathtt{idle}) \\ v & : & \mathtt{id} = 0,\ \mathtt{set} = 1,\ \mathtt{active}[0] = 3,\ \mathtt{active}[1] = 2,\ \mathtt{active}[2] = 0 \\ Z & : & x_0 = 2.5,\ x_1 = 3,\ x_2 = 4 \end{array}$$

The process swap swaps l_0 with l_2, and x_0 with x_2. Moreover, the value of the variable `id` is changed from 2 to 0, since $\mathtt{id} \in V_{\mathtt{proc\_id}}$, and the values of `active`[0] and `active`[2] are swapped. Applying $\mathit{swap}^{\mathtt{proc\_id}}_{1,2}$ to this state gives the following state:

$$\begin{array}{lcl} \vec{l} & : & (\mathtt{cs}, \mathtt{idle}, \mathtt{wait}) \\ v & : & \mathtt{id} = 0,\ \mathtt{set} = 1,\ \mathtt{active}[0] = 3,\ \mathtt{active}[1] = 0,\ \mathtt{active}[2] = 2 \\ Z & : & x_0 = 2.5,\ x_1 = 4,\ x_2 = 3 \end{array}$$

Note that this swap does not change the value of `id`, since the scalarset elements 1 and 2 are interchanged and `id` contains scalarset element 0.

[5]The soundness theorem has also been proved correct for the more general definition of the state swap function that appears in [55]. Thus, a definition of state swaps exists such that Theorem 4.8 holds without the need for Assumption 4.4.

4.4.2 A Preorder on Clocks

The zone semantics seems to render a straightforward comparison of clocks impossible, since there are in general many different clock valuations in a zone. If we assume, however, that the UPPAAL model resets its clocks to zero only and that the convex-hull over-approximation is not used, then the zones that are generated by the forward state space exploration satisfy the *diagonal property*. This property informally states that a zone never contains valuations on both sides of a diagonal. This implies that the individual clocks can always be ordered using the order in which they were reset. To formalize this, three binary relations on the set of clocks X parameterized by a zone Z are defined:

$$x \preccurlyeq_Z y \iff \forall_{\nu \in Z}\ \nu(x) \leq \nu(y) \tag{4.6}$$

$$x \approx_Z y \iff \forall_{\nu \in Z}\ \nu(x) = \nu(y) \tag{4.7}$$

$$x \prec_Z y \iff (x \preccurlyeq_Z y \wedge x \not\approx_Z y) \tag{4.8}$$

Clearly, the relation $\preccurlyeq_Z$ is reflexive and transitive and hence it is a preorder on the set of clocks. Totality of this preorder w.r.t. zones that are generated during the state space exploration follows from the diagonal property.

Lemma 4.9 (Diagonal property) *Consider the state space exploration algorithm described in Figure 6 of [16][6]. Assume that the clocks are reset to the value 0 only and that the convex-hull over-approximation is not used. Then for all states $(\vec{l}, v, Z)$ stored in the waiting and passed list during a run of the algorithm and for all clocks x and y it holds that either $x \prec_Z y$, $x \approx_Z y$ or $y \prec_Z x$.*

PROOF. We prove that the diagonal property holds for the arbitrary clocks x and y by an inductive argument. Consider the initial zone, which contains only one clock valuation. It is clear that the diagonal property holds for such a zone. Before we prove the induction step we observe that $x \approx_Z y$, $x \prec_Z y$, and $y \prec_Z x$ are mutually exclusive: if one holds then the remaining two do not hold. Now consider a zone Z that satisfies the diagonal property. If the convex-hull over-approximation is not used, then the three operations on zones that are used during state space exploration are the following [5]:

1. Intersection with zone Z'. This results in a zone $Z'' \subseteq Z$. Now assume that $x \prec_Z y$, which means that $\nu(x) \leq \nu(y)$ for all $\nu \in Z$, and that a $\nu \in Z$ exists such that $\nu(x) \neq \nu(y)$. Clearly, for all $\nu'' \in Z''$ still holds that $\nu''(x) \leq \nu''(y)$. Next, we distinguish two cases: First, a $\nu'' \in Z''$ exists such that $\nu''(x) \neq \nu''(y)$. Then clearly $x \prec_{Z''} y$. Second, such a ν'' does not exist. Then $\nu''(x) = \nu''(y)$ for all $\nu'' \in Z''$ and thus $x \approx_{Z''} y$. It is straightforward to see that $x \approx_Z y$ implies that $x \approx_{Z''} y$. Thus, the diagonal property also holds for Z''.

[6] Essentially, this is a UPPAAL tailored instance of the algorithm in Figure 4.1 of the present paper.

2. Resetting clock x in zone Z. This results in the zone $Z' = \{\nu[x := 0] \mid \nu \in Z\}$, where $\nu[x := 0](u) = \nu(u)$ if $u \neq x$, and $\nu[x := 0](x) = 0$. We distinguish two cases:

   - $u \sim_Z w$, where $u \neq x$ and $w \neq x$. Thus, the clock valuations in Z' have not changed for u and w, and clearly $u \sim_{Z'} w$.
   - $x \sim_Z y$. If there exists a $\nu' \in Z'$ such that $\nu'(y) > 0$, then $x \prec_{Z'} y$ by definition, since $\nu'(x) = 0$ and $\nu'(y) \geq 0$ for all $\nu' \in Z'$. Otherwise, $\nu'(x) = \nu'(y) = 0$ for all $\nu' \in Z'$, and hence $x \approx_{Z'} y$. (Note that resetting clocks to values greater than zero destroys the property in this case.)

   Thus, the diagonal property also holds for Z'.

3. Time elapse. This removes the upper bounds on the individual clocks: $Z' = \{\nu + \delta \mid \nu \in Z \wedge \delta \in \mathbb{R}_+\}$, where $(\nu + \delta)(x) = \nu(x) + \delta$ for all $x \in X$. Now assume that $x \prec_Z y$, which means that $\nu(x) \leq \nu(y)$ for all $\nu \in Z$, and that a $\nu \in Z$ exists such that $\nu(x) \neq \nu(y)$. First, consider a $(\nu + \delta) \in Z'$. Clearly, $(\nu + \delta)(x) \leq (\nu + \delta)(y)$, since $\nu(x) \leq \nu(y)$. Second, consider the $\nu \in Z$ such that $\nu(x) \neq \nu(y)$. By definition, $(\nu + \delta) \in Z'$ for any δ. Clearly, $(\nu + \delta)(x) \neq (\nu + \delta)(y)$. Therefore, $x \prec_{Z'} y$. Now assume that $x \approx_Z y$, which means that $\nu(x) = \nu(y)$ for all $\nu \in Z$. By definition, $(\nu + \delta)(x) = (\nu + \delta)(y)$, and clearly $x \approx_{Z'} y$. Thus, the diagonal property also holds for Z'.

(Note that the convex-hull over-approximation uses some kind of union of zones which clearly does not preserve the diagonal property.) This proves that every zone generated during the state space exploration satisfies the diagonal property. ■

The clock parts of two substates can be compared using a lexicographical preorder that is based on the $\preccurlyeq_Z$ preorder.

Definition 4.10 (Clock preorder) *Let $\vec{c}_1$ and $\vec{c}_2$ be two clock index vectors with length k, and let q be a state with zone Z. We say that $\vec{c}_1 <_q \vec{c}_2$ if and only if*

$$\exists_{0 \leq i < k} \ \left(\rho^{-1}([\vec{c}_1]_i) \prec_Z \rho^{-1}([\vec{c}_2]_i) \ \wedge \ \forall_{0 \leq j < i} \ \left(\rho^{-1}([\vec{c}_1]_j) \approx_Z \rho^{-1}([\vec{c}_2]_j)\right)\right)$$

The non-strict version of the clock order is defined as usual: $\vec{c}_1 \leq_q \vec{c}_2$ if and only if $\vec{c}_1 <_q \vec{c}_2$ or $\rho^{-1}([\vec{c}_1]_j) \approx_Z \rho^{-1}([\vec{c}_2]_j)$ for all $0 \leq j < k$.

Lemma 4.11 *If the clocks in the model are reset to zero only and the convex hull over-approximation is not used, then the relation on clock index vectors of equal length as defined in Definition 4.10 is a total preorder.*

PROOF. Straightforward, since $\preccurlyeq_Z$ is a total preorder on the set of clocks under the mentioned premises (see Lemma 4.9). ■

The next lemma relates the clock preorder that is defined for zones to the projections of the state to the individual clock valuations.

Lemma 4.12 *Let $\vec{c}_1$ and $\vec{c}_2$ be two clock index vectors with length k and let q be a state with zone Z. If some $\nu \in Z$ exists such that $[\![\vec{c}_1]\!]_\nu < [\![\vec{c}_2]\!]_\nu$, then $\vec{c}_1 <_q \vec{c}_2$.*

PROOF. Assume that a $\nu \in Z$ exists such that $[\![\vec{c}_1]\!]_\nu < [\![\vec{c}_2]\!]_\nu$. From Equation 4.5 we know that a $0 \leq j < k$ exists such that $\nu(\rho^{-1}([\vec{c}_1]_j)) < \nu(\rho^{-1}([\vec{c}_2]_j))$ and $\nu(\rho^{-1}([\vec{c}_1]_i)) = \nu(\rho^{-1}([\vec{c}_2]_i))$ for all $0 \leq i < j$. Now suppose that $\vec{c}_1 \not<_q \vec{c}_2$. Since the preorder is total (see Lemma 4.11), we consider the two remaining possibilities. First, suppose that $\vec{c}_1 =_q \vec{c}_2$. By Definition 4.10 and Equations 4.6–4.8: for all $\nu \in Z$ must hold that $\nu(\rho^{-1}([\vec{c}_1]_j)) = \nu(\rho^{-1}([\vec{c}_2]_j))$ for all $0 \leq j < k$. This clearly does not hold, and from this contradiction we can conclude that $\vec{c}_1 \neq_q \vec{c}_2$. Second, suppose that $\vec{c}_1 >_q \vec{c}_2$. By Definition 4.10 and Equations 4.6–4.8: for all $\nu \in Z$ must hold that $\nu(\rho^{-1}([\vec{c}_1]_j)) \geq \nu(\rho^{-1}([\vec{c}_2]_j))$ for all $0 \leq j < k$. This clearly also does not hold, and we conclude that $\vec{c}_1 \not>_q \vec{c}_2$. Thus, $\vec{c}_1 <_q \vec{c}_2$. ■

In the next subsection we define a total preorder on substates and use the state swaps to compute the representative of a symmetry class, which under certain assumptions is canonical.

4.4.3 Computation of Representatives

A comparison between the state contributions of different scalarset elements is defined as follows.

Definition 4.13 (Substate preorder) *Consider substate$(\alpha, i) = (\vec{l}_1, \vec{v}_1, \vec{c}_1)$, and also substate$(\alpha, j) = (\vec{l}_2, \vec{v}_2, \vec{c}_2)$, and let q be a state. Then substate$(\alpha, i) <_q$ substate(α, j) iff*

- $[\![\vec{l}_1]\!]_q < [\![\vec{l}_2]\!]_q$ *or*
- $[\![\vec{l}_1]\!]_q = [\![\vec{l}_2]\!]_q$ *and* $[\![\vec{v}_1]\!]_q < [\![\vec{v}_2]\!]_q$, *or*
- $[\![\vec{l}_1]\!]_q = [\![\vec{l}_2]\!]_q$ *and* $[\![\vec{v}_1]\!]_q = [\![\vec{v}_2]\!]_q$ *and* $\vec{c}_1 <_q \vec{c}_2$.

The non-strict version is defined as usual: substate$(\alpha, i) \leq_q$ substate(α, j) if and only if substate$(\alpha, i) <_q$ substate(α, j) or $[\![\vec{l}_1]\!]_q = [\![\vec{l}_2]\!]_q \wedge [\![\vec{v}_1]\!]_q = [\![\vec{v}_2]\!]_q \wedge \vec{c}_1 =_q \vec{c}_2$.

Lemma 4.14 *If the clocks in the model are reset to zero only and the convex-hull over-approximation is not used, then the relation as defined in Definition 4.13 is a total preorder on the substates of a scalarset.*

PROOF. Straightforward since the preorders on the three components of the substate are total under the mentioned assumptions. ■

The next lemma states how the substate preorder is affected by state swaps.

Lemma 4.15 *Let $q' = swap^{\alpha}_{i,j}(q)$, and let $\pi(i) = j$, $\pi(j) = i$, and $\pi(k) = k$ for all $k \neq i, j$. Assume that substate$(\beta, m) \sim_q$ substate(β, n), where $\sim \in \{<, =, >\}$.*

- *If $\alpha \neq \beta$, then substate$(\beta, m) \sim_{q'}$ substate(β, n).*
- *If $\alpha = \beta$, then substate$(\beta, \pi(m)) \sim_{q'}$ substate$(\beta, \pi(n))$.*

PROOF. The first case can easily be proved using Lemma 4.5. The state swap of α does not affect the parts of the state that are relevant for the substates of β. Therefore, the order of the elements of β is not disturbed. For the second case assume that $\alpha = \beta$ and let $q = (\vec{l}, v, Z)$, $q' = (\vec{l'}, v', Z')$, $s_1 =$ *substate*$(\beta, m) = (\vec{l}_1, \vec{v}_1, \vec{c}_1)$, and $s_2 =$ *substate*$(\beta, n) = (\vec{l}_2, \vec{v}_2, \vec{c}_2)$.

- Suppose that $m = n$. Since *substate*$(\beta, k) =_q$ *substate*(β, k) for all states q and for all $\beta \in \Omega$ and $k \in \beta$ the lemma clearly holds.
- Suppose that i and j are not equal to m or n. Thus, $\pi(m) = m$ and $\pi(n) = n$. This case can easily be proved using Lemma 4.5 and Definition 4.7: the state swap does not affect the projections to the substates of m and n.
- Suppose that $m = i$ and $n \neq i, j$. Thus, $\pi(m) = j$ and $\pi(n) = n$. Let $s_3 =$ *substate*$(\beta, j) = (\vec{l}_3, \vec{v}_3, \vec{c}_3)$. We prove that $[\![\vec{l}_3]\!]_{q'} = [\![\vec{l}_1]\!]_q$, $[\![\vec{v}_3]\!]_{q'} = [\![\vec{v}_1]\!]_q$, and that if $\vec{c}_1 \sim_q \vec{c}_2$, then $\vec{c}_3 \sim_{q'} \vec{c}_2$, where $\sim \in \{<, =, >\}$. This proves (see Definition 4.13) that $s_3 \sim_{q'} s_2$.

  1. Let $\vec{l}_1 = (l^0_1, l^1_1, \ldots, l^n_1)$ and let $\vec{l}_3 = (l^0_3, l^1_3, \ldots, l^n_3)$. By Definition 4.7 we see that $[\vec{l'}]_k = [\vec{l}]_k$ if $k \notin \vec{l}_1$ and $k \notin \vec{l}_2$. Moreover, the definition has the effect that the values of the entries at indices l^i_1 and l^i_3 are swapped for all $0 \leq i \leq n$. Clearly, $[\![\vec{l}_3]\!]_{q'} = [\![\vec{l}_1]\!]_q$.
  2. Since we assumed that V_α only contains global non-array variables (see Assumption 4.4 and the second assumption at the beginning of Section 4.4.1), we can use the same argument as in the previous item to prove $[\![\vec{v}_3]\!]_{q'} = [\![\vec{v}_1]\!]_q$.
  3. Assume that $\vec{c}_1 <_q \vec{c}_2$ and let $\vec{c}_1 = (c^0_1, c^1_1 \ldots, c^k_1)$, and let $\vec{c}_2 = (c^0_2, c^1_2, \ldots, c^k_2)$. By Definition 4.10 we know that some $0 \leq f \leq k$ exists such that:

$$\rho^{-1}(c^f_1) \prec_Z \rho^{-1}(c^f_2) \tag{4.9}$$

$$\forall_{0 \leq e < f}\ \rho^{-1}(c^e_1) \approx_Z \rho^{-1}(c^e_2) \tag{4.10}$$

  This means by definition of the $\approx_Z$ and $\prec_Z$ relations that

$$\exists_{\nu \in Z}\ \nu(\rho^{-1}(c^f_1)) < \nu(\rho^{-1}(c^f_2)) \tag{4.11}$$

$$\forall_{0 \leq e < f}\ \forall_{\nu \in Z}\ \nu(\rho^{-1}(c^e_1)) = \nu(\rho^{-1}(c^e_2)) \tag{4.12}$$

  Next, we apply Definition 4.7 which swaps the values of $\rho^{-1}(c^l_1)$ and $\rho^{-1}(c^l_3)$ for all $0 \leq l \leq n$ and lets the other clocks unchanged. Since

substates are disjoint (see Lemma 4.5) we know that the clocks in $\vec{c}_2$ keep their values: $s(\nu)(\rho^{-1}(c_2^l)) = \nu(\rho^{-1}(c_2^l))$ for all $0 \leq l \leq n$. Furthermore, $s(\nu)(\rho^{-1}(c_1^l)) = \nu(\rho^{-1}(c_3^l))$ for all $v \in Z$ and $0 \leq l \leq n$. Combination with the previous equations gives us that:

$$\exists_{\nu \in Z'}\ \nu(\rho^{-1}(c_3^f)) < \nu(\rho^{-1}(c_2^f)) \tag{4.13}$$

$$\forall_{0 \leq e < f}\ \forall_{\nu \in Z'}\ \nu(\rho^{-1}(c_3^e)) = \nu(\rho^{-1}(c_2^e)) \tag{4.14}$$

By definition of the $\approx$ and $\prec$ relations:

$$\rho^{-1}(c_3^f) \prec_{Z'} \rho^{-1}(c_2^f) \tag{4.15}$$

$$\forall_{0 \leq e < f}\ \rho^{-1}(c_3^e) \approx_{Z'} \rho^{-1}(c_2^e) \tag{4.16}$$

By Definition 4.10 we can conclude that $\vec{c}_3 <_{q'} \vec{c}_2$. The case for $\vec{c}_1 =_q \vec{c}_2$ is similar.

- Suppose that $m = i$ and $n = j$. Thus, $\pi(m) = j$ and $\pi(n) = i$. With a similar argument as in the previous item we can prove that $[\![\vec{l}_1]\!]_{q'} = [\![\vec{l}_2]\!]_q$, $[\![\vec{l}_2]\!]_{q'} = [\![\vec{l}_1]\!]_q$, $[\![\vec{v}_1]\!]_{q'} = [\![\vec{v}_2]\!]_q$, $[\![\vec{v}_2]\!]_{q'} = [\![\vec{v}_1]\!]_q$, and if $\vec{c}_1 \sim_q \vec{c}_2$, then $\vec{c}_2 \sim_{q'} \vec{c}_1$, where $\sim \in \{<, =, >\}$. Hence, $s_2 \sim_{q'} s_1$.

■

We minimize the state by sorting the substate contributions of each scalarset according to the substate preorder of Definition 4.13. To this end, we apply a variation of the bubble-sort algorithm, see Figure 4.3. It is clear that this representative computation satisfies Equation 4.1 which ensures soundness, since states are transformed using the state swaps only, which are automorphisms by Theorem 4.8.

```
(1)   for all α ∈ Ω do
(2)       for i = 1 to |α| do
(3)           for j = |α| − 1 to i do
(4)               if substate(α, j) <_q substate(α, j − 1) then
(5)                   q := swap^α_{j−1,j}(q)
(6)               {substate(α, j − 1) ≤_q substate(α, m), j ≤ m < |α|}
(7)           od
(8)           {substate(α, 0) ≤_q ··· ≤_q substate(α, i − 1)}
(9)       od
(10)      {m ≤ n ⇒ substate(α, m) ≤_q substate(α, n)}
(11)  od
```

Figure 4.3: Minimization of state q using the bubble-sort algorithm. The size of scalarset type α is denoted by $|\alpha|$. Lines 6, 8 and 10 show the loop invariants.

The following theorem states the main technical contribution of our work. Informally, it means that the detected symmetries are optimally used.

Theorem 4.16 (Canonical representative) *If Assumptions 4.4 and 4.6 are true, the convex hull over-approximation is not used, there are no variables of a scalarset type ($V_\alpha = \emptyset$) and the clocks in the model are reset to zero only, then the representative function θ as computed by the algorithm in Figure 4.3 is canonical.*

PROOF. The loop invariants can easily be proved using the fact that the preorder on substates is total (Lemma 4.14) and the fact that swapping two elements of a scalarset only has an effect on those two elements, i.e., the relations between other substates are not disturbed (Lemma 4.15).

Next, we prove that the algorithm computes a canonical representative, i.e., θ satisfies $q \approx q' \Rightarrow \theta(q) = \theta(q')$. Suppose that $q \approx q'$, i.e., a sequence of state swaps exists that transforms q into q'. Now consider $\theta(q) = (\vec{l}, v, Z)$ and $\theta(q') = (\vec{l'}, v', Z')$ (clearly, $\theta(q) \approx \theta(q')$ by Equation 4.1) and assume that they are different.

1. Assume that $\vec{l} \neq \vec{l'}$, more precisely, $[\vec{l}]_k \neq [\vec{l'}]_k$, and $[\vec{l}]_j = [\vec{l'}]_j$ for all $1 \leq j < k$. Without loss of generality we can also assume that $[\vec{l}]_k > [\vec{l'}]_k$. Clearly, some scalarset α and $i \in \alpha$ exist such that $k \in \vec{l_i}$, where $\vec{l_i} = [\textit{substate}(\alpha, i)]_0$ since otherwise entry k cannot be swapped and as a result $\theta(q) \not\approx \theta(q')$ which we assumed. Moreover, exactly one such a combination exists, since substates are disjoint according to Lemma 4.5. Say that $\vec{l_i} = (l_i^0, l_i^1, \ldots, l_i^n)$ and that $l_i^h = k$. By Definition 4.3 we know that $l_i^g < l_i^h$ for all $0 \leq g < h$. Combination with the fact that $[\vec{l}]_j = [\vec{l'}]_j$ for all $1 \leq j < k$ gives us then that $[\![\vec{l_i}]\!]_{\theta(q)} > [\![\vec{l_i}]\!]_{\theta(q')}$.
   Since $\theta(q) \approx \theta(q')$ it is possible to transform $\theta(q)$ into $\theta(q')$ by state swaps. By Definition 4.7 and the fact that substates are disjoint (see Lemma 4.5), the i-th element of α can only be replaced by the following elements of α as a result of such a transformation:

$$J = \{\, j \mid [\![\vec{l_j}]\!]_{\theta(q)} = [\![\vec{l_i}]\!]_{\theta(q')} \text{ where } \vec{l_j} = [\textit{substate}(\alpha, j)]_0 \wedge 0 \leq j < |\alpha| \,\}$$

   Clearly, $J \neq \emptyset$ since that would mean that $\theta(q) \not\approx \theta(q')$. Now we show that a $j \in J$ must exist such that $j > i$. Therefore, assume that $j \leq i$ for all $j \in J$, and consider the following set of location vector indices which are exactly those indices whose entries in $\vec{l}$ can replace the value $[\vec{l}]_k$ as a result of the transformation that proves that $\theta(q) \approx \theta(q')$:

$$G = \{\, [\vec{l_j}]_h \mid \text{where } \vec{l_j} = [\textit{substate}(\alpha, j)]_0 \text{ and } j \in J \,\}$$

   By Assumption 4.6 and the assumption that $j \leq i$ (and $j \neq i$) for all $j \in J$ we know that $g < k$ for all $g \in G$. Thus, $[\vec{l}]_g = [\vec{l'}]_g$ for all $g \in G$ and $[\vec{l}]_k \neq [\vec{l'}]_k$. Clearly, $\vec{l}$ can never be transformed into $\vec{l'}$. This contradicts our assumption that $\theta(q) \approx \theta(q')$, and therefore we can conclude that a $j \in J$ exists such that $j > i$. Now we fix this $j > i$ and have that

$[\![\vec{l_j}]\!]_{\theta(q)} = [\![\vec{l_i}]\!]_{\theta(q')}$. Above we have shown that $[\![\vec{l_i}]\!]_{\theta(q)} > [\![\vec{l_i}]\!]_{\theta(q')}$. Combination gives us that $i < j$ and $[\![\vec{l_i}]\!]_{\theta(q)} > [\![\vec{l_j}]\!]_{\theta(q)}$. In other words, $i < j$ and *substate*$(\alpha, i) >_{\theta(q)}$ *substate*(α, j), which clearly contradicts the loop invariant in line 10 of the algorithm in Figure 4.3. Therefore, $\vec{l} = \vec{l'}$.

2. Assume that $\vec{l} = \vec{l'} \wedge v \neq v'$. Since we assumed that $V_\alpha = \emptyset$ we know that the difference between v and v' is in the projections to some substate. Therefore, the proof is the same as in the previous item.

3. Assume that $\vec{l} = \vec{l'} \wedge \vec{v} = \vec{v'} \wedge Z \neq Z'$. This means that a $\nu \in Z$ exists such that $\nu \notin Z'$. Moreover, since $\theta(q) \approx \theta(q')$, a $\nu' \in Z'$ exists such that ν can be transformed into ν' by the state swaps. Let us consider this ν and this ν'. By assumption, a clock index k exists such that $\nu(\rho^{-1}(k)) \neq \nu'(\rho^{-1}(k))$ and $\nu(\rho^{-1}(i)) = \nu'(\rho^{-1}(i))$ for all $0 \leq i \leq k$. Without loss of generality we can assume that $\nu(\rho^{-1}(k)) > \nu'(\rho^{-1}(k))$, and that $k \in \vec{c_i} = [$*substate*$(\alpha, i)]_2$. Thus, $[\![\vec{c_i}]\!]_\nu > [\![\vec{c_i}]\!]_{\nu'}$. With a similar argument as in the first item we can prove that a $j > i$ exists such that $[\![\vec{c_j}]\!]_\nu = [\![\vec{c_i}]\!]_{\nu'}$, where $\vec{c_j} = [$*substate*$(\alpha, j)]_2$.
Combination gives us that $i < j$ and $[\![\vec{c_i}]\!]_\nu > [\![\vec{c_j}]\!]_\nu$. Applying Lemma 4.12 gives us then that $i < j$ and $\vec{c_i} >_{\theta(q)} \vec{c_j}$, which clearly contradicts the loop invariant in line 10 of the algorithm in Figure 4.3. Therefore, $Z = Z'$.

■

Due to the presence of the global variable `id`, which has scalarset type, our model of Fischer's protocol does not satisfy the conditions of the theorem above. And indeed the representative function θ as computed by the algorithm in Figure 4.3 is not fully canonical for this model. This is due to the fact that two processes can take the transition to location `wait` at the same moment. The projections to the resulting substates of the processes then are equal, but the value of `id` depends on the order of arrival. Our algorithm cannot distinguish these two different states. We claim, however, that the implementation does compute a canonical representative, since it also considers the V_α variables for the decision whether to swap two scalarset elements.

4.5 Experimental Results

This section presents and discusses experimental data that has been obtained with the UPPAAL prototype. The measurements were done using the tool *memtime*, for which a link can be found at the UPPAAL website.

In order to demonstrate the effectiveness of symmetry reduction, the resource requirements for checking the correctness of Fischer's mutual exclusion protocol were measured as a function of the number of processes for both regular UPPAAL and the prototype, see Figure 4.4. A conservative extrapolation of the data shows

that the verification of the protocol for 20 processes without symmetry reduction would take 115 days and 1000 GB of memory, whereas this verification can be done within approximately one second using less than 10 MB of memory with symmetry reduction.

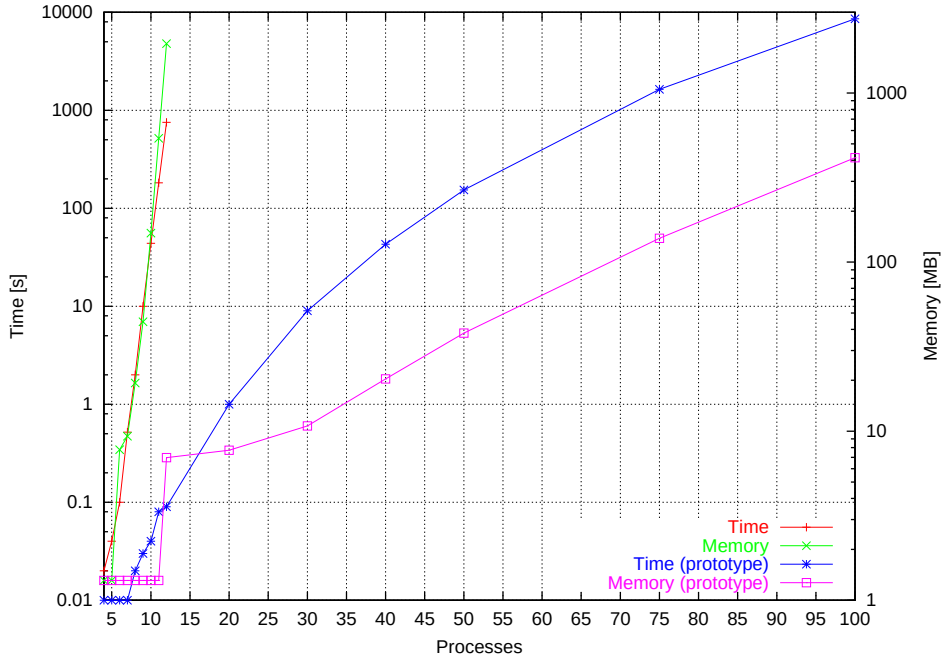

Figure 4.4: Run-time data for Fischer's mutual exclusion protocol showing the enormous gain of symmetry reduction. The step in the graph of the memory usage is probably due to the the fact that UPPAAL allocates memory in chunks of a few megabyte at a time.

Similar results have been obtained for the CSMA/CD protocol ([99, 107]) and for the timeout task of a distributed agreement algorithm which is described in [10]. To be more precise, regular UPPAAL's limit for the CSMA/CD protocol is approximately ten processes, while the prototype can easily handle fifty processes. Similarly, the prototype can easily handle thirty processes for the model of the timeout task, whereas regular UPPAAL can only handle six processes.

Besides the three models discussed above, we also investigated the gain of symmetry reduction for two more complex models. First, we measured the gain for the previously mentioned agreement algorithm, of which we are unable to verify an interesting instance even with symmetry reduction due to the size of the state space. Nevertheless, symmetry reduction showed a very significant improvement for less interesting instances of the algorithm (only two symmetric processes). Second, we measured the gain for a model of Bang & Olufsen's audio/video protocol, which is described in [53]. This paper describes how UPPAAL is used to find an error in the protocol, and it describes the verification of the corrected protocol for two

(symmetric) senders. (It is interesting to note that analysis of the corrected version of the protocol in [100] reveiled another error.) Naturally, we added another sender – verification of the model for three senders was impossible at the time of the first verification – and we found another error, whose source and implications we are investigating at the time of this writing. Table 4.1 shows run-time data for these models.

Table 4.1: Comparing the time and memory consumption of the prototype with the regular tool for the agreement algorithm (where the message delay varies) and for Bang & Olufsen's audio/video protocol with two and three senders. Three verification runs were measured for each model and the best one w.r.t. time is shown.

Model	**Time [s]**		**Memory [MB]**	
	No red.	Red.	No red.	Red.
Agreement (0)	1	3	33	45
Agreement (1)	21	16	294	180
Agreement (2)	80	23	905	245
Agreement (3)	231	32	2126	321
B&O (2)	2	1	16	10
B&O (3)	265	36	1109	181

4.6 Conclusions

The results we obtained with our prototype are clearly quite promising: with relatively limited changes/extensions of the UPPAAL code we obtain a rather drastic improvement of performance for systems with symmetry that can be expressed using scalarsets.

An obvious next step is to do experiments concerning profiling where computation time is spent, and in particular how much time is spent on computing representatives. In the tool Design/CPN [66, 71, 47] (where symmetry reduction is a main reduction mechanism) there have been interesting prototype experiments with an implementation in which the (expensive) computations of representatives were launched as tasks to be solved in parallel with the main exploration algorithm.

As noted before, due to the presence of the global variable `id`, which has scalarset type, our model of Fischer's protocol does not satisfy the conditions of Theorem 4.16. We claim, however, that the implementation does compute a canonical representative, since it also considers the V_α variables for the decision whether to swap two scalarset elements. Nevertheless, of course, it remains an interesting topic for future research to optimize the representative function for timed automata models that do not satisfy the restrictions of Theorem 4.16.

In this paper, we have exploited symmetries to statically derive bisimulations

and (efficient) representative functions from system descriptions. A complementary static analysis technique for deriving bisimulations and representative functions is the *dead variable reduction* technique described in the PhD thesis of Karen Yorav [106]. In Yorav's terminology, a variable v is *used* in a transition $l \overset{g,a,up}{\longrightarrow} l'$ if v appears in g or in the right hand side of an assignment in up. Variable v is *defined* in the transition if it is in the left hand side of an assignment in up. Notice that in an assignment "$v := v + 1$" v is first used, and then it is defined. A variable v is said to be *dead* at location l if on every execution path from l, v is defined before it is used, or is never used at all. Clearly, states that only differ on the values of dead variables are bisimilar, and any function that assigns a fixed value to these variables will give us a canonical representative function. An example of a dead variable is the global variable `id` in Fischer's protocol, of which the value does not matter for locations in which none of the components is in its waiting location. Dead variable reduction is closely related to the static guard analysis technique for timed automata as described in [13]. It would interesting to implement dead variable reduction in UPPAAL and to investigate the resulting speedup on some benchmark examples.

The scalarset approach that we follow in this paper only allows one to express total symmetries. An obvious direction for future research will be to study how other types of symmetry (for instance as we see it in a token ring) can be exploited.

Chapter 5

Model Checker Aided Design of a Controller for a Wafer Scanner

MARTIJN HENDRIKS

BAREND VAN DEN NIEUWELAAR

FRITS VAANDRAGER

Abstract. For a case-study of a wafer scanner from the semiconductor industry it is shown how model checking techniques can be used to compute (i) a simple yet optimal deadlock avoidance policy, and (ii) an infinite schedule that optimizes throughput. Deadlock avoidance is studied based on a simple finite state model using SMV, and for throughput analysis a more detailed timed automaton model has been constructed and analyzed using the UPPAAL tool. The SMV and UPPAAL models are formally related through the notion of a stuttering bisimulation. The results were obtained within two weeks, which confirms once more that model checking techniques may help to improve the design process of realistic, industrial systems. Methodologically, the case study is interesting since two models were used to obtain results that could not have been obtained using only a single model.

5.1 Introduction

Scheduling and resource allocation problems occur in many different domains, for instance (1) scheduling of production lines in factories to optimize costs and delays, (2) scheduling of computer programs in (real-time) operating systems to meet deadline constraints, (3) scheduling of instructions inside a processor with a bounded number of registers and processing units, (4) scheduling of trains (or airplanes) over limited quantities of railway tracks and crossroads, and (5) mission planning for autonomous robots on spacecrafts. Typically, in each of these domain problems are solved using different approaches and mathematical tools. The EU IST project AMETIST envisages a unifying framework for time dependent behavior and dynamic resource allocation that crosses the boundaries of application domains.

In the AMETIST approach, components of a system are modeled as *dynamical systems* with a state space and a well-defined dynamics. All that can happen in a system is expressed in terms of *behaviors* that can be generated by the dynamical systems; these constitute the semantics of the problem. Verification, optimization, synthesis and other design activities explore and modify system structure so that the resulting behaviors are correct, optimal, etc. Preferably, the limitations of currently

This chapter is a literal copy of [60].

known computational solutions should not influence modeling too much: only after the semantics of a problem is properly understood, abstractions and specialization due to computational considerations can intervene. In such situations, the soundness of abstractions should ideally also be proved, either via deductive verification or model checking. AMETIST aims to extend this approach, which underlies the successful domain of *formal verification*, to resource allocation, scheduling and other time-related problems. The present paper serves as an illustration of this methodology.

A major concern in the design of controllers for many resource allocation systems (RASs) is *deadlock*, a permanently blocking condition. There are three general ways of handling deadlock: (i) deadlock prevention, (ii) deadlock detection and resolution, and (iii) deadlock avoidance. Deadlock prevention restricts the system in such a way that deadlock is a priori impossible. As a consequence, performance may be unnecessarily low. Deadlock detection and resolution, on the other hand, is not restrictive at all and detects and resolves a deadlock at run-time. This, however, may be very expensive. Deadlock avoidance achieves a middle ground; it dynamically chooses the control actions to avoid the occurrence of deadlock. In this paper, we show how a least restrictive deadlock avoidance policy (DAP) for the wafer scanner can be easily computed using SMV, a model checker for finite automata. This DAP can be represented by a very short predicate over the states of the wafer scanner, which can be used by the controller for the wafer scanner. In addition, we use the timed automaton tool UPPAAL to define a refined model that adds timing constraints to address the issue of throughput optimization. We relate the UPPAAL model to the SMV model via the concept of *stuttering bisimulation* introduced by Browne, Clarke and Grumberg [31]. Since stuttering bisimulation preserves validity of CTL formulas (without nexttime operator), all properties (and in particular the DAP) that we established for the untimed model using SMV, carry over to the UPPAAL model. It is not possible to compute the least restrictive DAP directly for the UPPAAL model since (a) UPPAAL does not support full CTL, and (b) the state space of the UPPAAL model is so big that it cannot be fully explored. Using heuristics, however, we are able to use the UPPAAL model checker to find an infinite schedule that optimizes throughput.

Contribution. The main contribution of our paper is a uniform, model based approach to deal with both deadlock avoidance and throughput optimization. We present a case study in which for each of these two problems a model is constructed and a solution is computed with a model checker. The two models are related formally through the notion of a stuttering bisimulation. We are not aware of other work that addresses both deadlock avoidance and throughput optimization in (what essentially is) a single framework.

Our results were obtained within two weeks, and we believe that our method can be applied by engineers with a background in computer science after training of only a few days. This confirms that model checking may help to improve the design process of realistic, industrial systems. Our DAP computation approach is referred to in a patent application of ASML, which shows its significance for

industry. Methodologically, the case study is interesting since two models were used in combination to obtain results that could not have been obtained using only a single model. Our approach illustrates once more that building models that are just abstract enough for addressing a specific question, often provides a way to deal with the state space explosion problem. Probably, we could have carried out the complete analysis using a single tool, namely KRONOS [107], a model checker for timed automata that supports full Timed CTL. We decided to use UPPAAL because of its greater maturity, efficiency and user friendliness. In particular, the graphical user interface and simulator facilitated communication about our model with the ASML engineers. Since UPPAAL does not support full CTL model checking, we used SMV, which is also very mature and efficient, for the computation of the DAP.

Related work. Much research has been devoted to deadlock avoidance in RASs, see for instance [90, 94]. Discouraged by the NP-completeness of optimal deadlock avoidance for many RAS classes, see for instance [76], this kind of work generally focuses either on computation of suboptimal but polynomial DAPs or on optimal policies for very specific sub classes. Much of this work uses the Petri net formalism [86] for the modeling and analysis of RASs. Using these approaches, the deadlock avoidance problem from the present paper can be solved very easily.

In [52], the model checker SMV is used to construct a deadlock free controller by an iterative process. The parallel composition of the controller and the plant is checked against deadlock by SMV. If a deadlock state is found, then the controller is adjusted to exclude the counterexample and the verification is run again. Otherwise, the controller is deadlock free. The work presented in [105] deals with verification of several DAPs using SMV.

Papers in which model checking tools are used to solve scheduling problems include a case study in which a control schedule for a smart card personalization system is synthesized using the SMV model checker [50], a case study in which the UPPAAL model checker is used to find feasible schedules for a steel plant [49], a recent case study in which the UPPAAL model checker is used to find feasible schedules for lacquer production [15], and a case study by Niebert et al. [87] who used KRONOS [107] to synthesize infinite schedules with stationary throughput for a chemical batch plant. The present work is a follow-up on [30], which considers the same example and uses suboptimal deadlock avoidance heuristics to generate schedules that are not guaranteed to be optimal.

Outline. First, Section 5.2 informally presents the case study. Section 5.3 then presents the SMV model and shows two ways of obtaining an optimal DAP using SMV. In Section 5.4, a UPPAAL model of the wafer scanner is described, and infinite schedules which optimize throughput are computed. Also, we present a stuttering bisimulation that relates the UPPAAL model with the SMV model of Section 5.3. Finally, Section 5.5 draws some conclusions and gives directions for future work.

5.2 The EUV Machine

Lithographic machines, called *wafer scanners*, are used within the semiconductor industry to project chip designs on slices of silicon which are called wafers. A key performance characteristic of wafer scanners is throughput, i.e., the number of wafers that can be processed per time unit. For a typical recipe[1] it is desirable that the exposure operation (which uses the lens which is the most expensive part of the machine) is critical in optimal schedules. In order to maximize throughput, a controller should have a strategy that optimizes throughput in the absence of errors. Furthermore, a controller should be deadlock-free, since deadlock resolution is expensive. ASML aims at design-time verification of (key parts of) the control software for the wafer scanners that it develops, in order to prevent occurrence of errors while customers are using the machines.

Figure 5.1 schematically depicts a possible design of an *Extreme Ultra Violet machine* (EUV machine), which is a particular type of wafer scanner that is currently being developed by ASML. The inside of an EUV machine is kept vacuum as EUV light is absorbed by air. The wafer flow is presented in Figure 5.1. First,

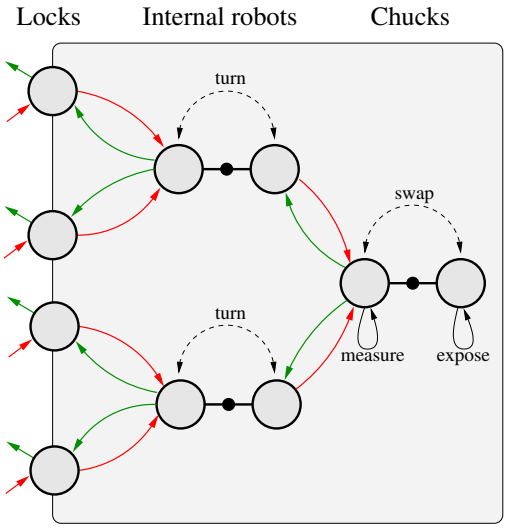

Figure 5.1: Wafer paths within the EUV machine.

the external track robot (which is not shown) puts a wafer in one of the four locks. This lock is depressurized, and then the wafer is picked up by one of the two internal robots. Each internal robot has two arms that can each hold a wafer and that are opposite to each other. The internal robot turns and puts the wafer on the closest chuck, which is in the so-called “measure position”. The wafer is measured and a chuck swap is performed. The chuck with the measured wafer now is in the “expose position” and the wafer is exposed. After another chuck swap, the

[1] The timing parameters of the production depend on the chips to be produced.

exposed wafer is picked up by one of the internal robots which turns and puts it in a depressurized lock. After the lock has been pressurized, the track robot removes the exposed wafer from the machine. Each wafer thus has a fixed recipe for its route: lock - internal robot - chuck - internal robot - lock. There is a choice which locks, internal robots and chucks are used by a wafer. An obvious question that arises is why we not let the unexposed wafers enter through the upper two locks and let the exposed wafers exit through the lower two locks. In that case there are no crossing material paths which means that we have deadlock prevention by construction. The answer is twofold. First, if locks are unidirectional then filling the machine from the initial, empty, state takes unnecessarily long. Second, if locks are unidirectional then the depressurization operation might become critical instead of the exposure, since depressurization takes more than twice as long as exposure in a typical wafer recipe. As noted above, this is undesirable. In Section 5.4, we will prove that indeed the exposure subsystem is critical in the design of Figure 5.1, and that restricting the wafer flow to prevent deadlock a priori lowers both the throughput and the utilization of the exposure subsystem.

A typical example of a deadlock situation in the EUV machine would be a state in which all four robot arms hold unprocessed wafers, and both chucks hold processed wafers. A controller for the EUV machine should ensure that no such deadlock situation can ever be reached. The problem of finding such a control strategy is commonly referred to as the deadlock avoidance problem. The EUV machine is a disjunctive RAS according to the taxonomy of [77]. Instead of the traditional Petri net or graph based approaches to solving the deadlock avoidance problem, we will show in the next section how it can be tackled using the SMV model checker.

5.3 Least Restrictive Deadlock Avoidance Policy

In this section, after a (very) brief introduction into SMV, we present our SMV model of the EUV machine, discuss how one can formalize the notion of deadlock as a temporal logic formula, and present the deadlock avoidance policy that we synthesized using SMV. The reader is referred to [37] and [83] for an extensive introduction into model checking and SMV.

5.3.1 SMV

In the approach supported by the SMV model checker, a system is modeled as a finite *transition system*, i.e. as a tuple $(S, s_{\text{init}}, \rightarrow)$ where S is a finite set of states, s_{init} is the initial state, and $\rightarrow \subseteq S \times S$ is the transition relation. We write $s \rightarrow s'$ instead of $(s, s') \in \rightarrow$. A state is defined as a valuation of a number of *state variables*. The value of state variable v in state s is denoted by $s(v)$. Furthermore, $s[v := c]$ denotes the state that is obtained by updating the value of v in state s to c. A *path* of a transition system is a sequence $s_0 s_1 s_2 \cdots$ such that for all i, $s_i \rightarrow s_{i+1}$.

A state is *reachable* if it occurs on some path that starts in s_{init}.

In SMV, specifications are described in *Computation Tree Logic (CTL)* which is a branching time temporal logic. Below some examples of CTL formulas are given, which should be sufficient to understand the present paper. The basic building blocks of CTL are *atomic formula*, which denote functions from the set of states to $\{true, false\}$. For instance, if v is a state variable, then $v = 2$ is an atomic formula, which denotes the function from states to $\{true, false\}$ that maps a state s to *true* iff $s(v) = 2$. In this case, we say state s *satisfies* formula $v = 2$, notation $s \models (v = 2)$. Every atomic formula is a *state formula*. State formulas can be combined with Boolean connectives and *path operators*. We show three path operators that are relevant for this paper. First, if ϕ is a state formula, then $\mathbf{AG}(\phi)$ also is a state formula. A state s satisfies $\mathbf{AG}(\phi)$, denoted by $s \models \mathbf{AG}(\phi)$, if for all paths $s_0 s_1 s_2 \ldots$ with $s = s_0$, and for all $i \geq 0$, $s_i \models \phi$. Second, if ϕ is a state formula, then $\mathbf{EF}(\phi)$ is also a state formula. We define $s \models \mathbf{EF}(\phi)$ if there exists a path $s_0 s_1 s_2 \ldots$ such that $s = s_0$ and $s_i \models \phi$, for some $i \geq 0$. Finally, if ϕ is a state formula, then $\mathbf{EG}(\phi)$ also is a state formula. We define $s \models \mathbf{EG}(\phi)$ if there exists a path $s_0 s_1 s_2 \ldots$ with $s = s_0$ such that for all $i \geq 0$, $s_i \models \phi$.

5.3.2 An SMV Model of the EUV Machine

The EUV machine can be modeled conveniently and concisely in SMV. In fact, the full code is displayed in Figure 5.2.

For each of the 10 positions in the machine our model contains a state variable: an array `l` of size 4 for the locks, a 2-dimensional array `rb` of size 2×2 for the robots, and an array `c` of size 2 for the chucks. These state variables can either take value `e` (*empty*), which means that the position is empty, value `r` (*red*), which means that the position is occupied by an unexposed wafer, or `g` (*green*), which means that the position is occupied by an exposed wafer. Initially, the machine is completely empty and all state variables have value `e`.

To model the system dynamics, i.e., the movement and exposure of wafers, we introduce 22 asynchronous processes, which are executed in an interleaving fashion:

- For each of the 4 locks `i` we have process `tl[i]`, which may either put an unexposed wafer in lock `i` if it is empty, or move an exposed wafer from the lock to the track robot. In the definition of process `tl[i]` we use an auxiliary function `entry_exit` that describes the state change that results from running this process.

- For each of the 16 pairs of positions `i`, `j` such that `i` is on the left of `j` and a wafer can move directly from `i` to `j` (or back), we introduce a process that takes care of moving unexposed wafers from `i` to `j`, and exposed wafers from `j` back to `i`. In the definition of these processes we use a function `move(lft,rgt)` that describes the state change that results from moving a wafer from `lft` to `rgt` or vice versa.

```
module entry_exit (p)
{
  if (p=e) next(p):=r;
  else if (p=g) next(p):=e;
}

module move (lft,rgt)
{
  if (lft=r && rgt=e){
    next(lft):=e;
    next(rgt):=r;
  }
  else if (lft=e && rgt = g){
    next(lft):=g;
    next(rgt):=e;
  }
}

module expose (p)
{
  if (p=r) next(p):=g;
}

module main ()
{
  -- state variables
  l : array 0..3 of {e,r,g};
  rb: array 0..1 of array 0..1 of {e,r,g};
  c : array 0..1 of {e,r,g};

  -- initialization
  for (i=0; i<4; i=i+1) init(l[i]):=e;
  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1) init(rb[i][j]):=e;
  for (i=0; i<2; i=i+1) init(c[i]):=e;

  -- system dynamics
  for (i=0; i<4; i=i+1)
    tl[i]: process entry_exit(l[i]);

  for (i=0; i<4; i=i+1)
    for (j=0; j<2; j=j+1)
      lr[i][j]:
        process move(l[i],rb[(i<2?0:1)][j]);

  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      for (k=0; k<2; k=k+1)
        rc[i][j][k]:
          process move(rb[i][j],c[k]);

  for (i=0; i<2; i=i+1)
    exp[i]: process expose(c[i]);
}
```

Figure 5.2: SMV model of EUV machine.

- For each of the 2 chucks `i` we introduce a process `exp[i]` that models exposure of the wafer. An auxiliary function `expose` describes the state change that results from exposing the wafer at position `p`: the value of the corre-

sponding state variable changes color from r (red) to g (green).

In the SMV model we abstract from the turning of internal robots. So a wafer can be picked up by both arms of an internal robot (possibly, the robot first has to turn). Similarly, the SMV model abstracts from chuck swaps and the measure operation. In Section 5.4, we present a more detailed model of the EUV machine in which we do not abstract from these aspects.

As it turns out, our SMV model has 57116 reachable states, which is close to the total number of states which equals $3^{10} = 59049$. An example of an unreachable state is one in which the machine is completely filled with exposed wafers. Transition systems of this size can very easily be handled by SMV and the computer hardware that is available today, so we expect that our approach can also be applied to considerably larger designs.

5.3.3 Defining Deadlock and Safety in SMV

Standard textbooks on operating systems, e.g. [97], state four conditions for deadlock in systems that consist of *processes* that compete for *resources*. The first three conditions concern the model itself and are necessary, and the fourth condition concerns the states of the model and is necessary and sufficient when the first three are met: (i) mutual exclusion: only one process may use a resource at a time, (ii) hold and wait: a process may hold allocated resources while awaiting assignment of others, (iii) no preemption: no resource can be forcibly removed from a process that is holding it, and (iv) circular wait: a closed chain of processes exists such that each process holds at least one resource needed by the next resource in the chain.

In the EUV machine, the wafers are modeled as the processes and they compete for the positions in the machine that constitute the resources. The model of the EUV machine satisfies the first three conditions for deadlock. The fourth condition, which thus is necessary and sufficient for deadlock, can be formalized with help from a *needs* function, that specifies for each wafer the set of positions it may move to. Let P denote the set of positions in the EUV machine. For $p \in P$ and $c \in \{\mathtt{r}, \mathtt{g}\}$, we define $\mathit{needs}(p, c) \subseteq P$ to be the set of positions (different from p) to which a wafer with color c at position p may move next. In particular, if p is a chuck, then $\mathit{needs}(p, \mathtt{r}) = \mathit{needs}(p, \mathtt{g}) = R$, where R is the set of positions of the internal robots. If s is a state and p a position then we use $\mathit{needs}^s(p)$ as an abbreviation for $\mathit{needs}(p, s(p))$. The circular wait property can now be defined as follows.

Definition 5.1 (Circular wait) *A state s has a* circular wait *in $Q \subseteq P$ iff $s(q) \neq \mathtt{e} \wedge \emptyset \neq \mathit{needs}^s(q) \subseteq Q \neq \emptyset$ for all $q \in Q$.*

It is not possible to directly formulate the circular wait property in terms of CTL, so some encoding is required. The basic idea is that the machine has a circular wait in a subset Q of positions iff the wafers in Q will never be able to move again. Observe that if in our model a transition $s \rightarrow s'$ moves a wafer from place p

to place p', then p is empty in s'. Thus, the property that some wafer cannot move anymore can be formalized in CTL as follows.

Definition 5.2 (Jam) *A position p is* jammed *in state s iff $s \models \mathbf{AG}(p \neq \mathtt{e})$. A state s is* jammed *iff some position is jammed in s.*

Proposition 5.5 below asserts the equivalence of the circular wait and jammed properties, thereby providing us with a way to express deadlocks in CTL. In order to prove the proposition, we need two technical lemmas stating that (a) circular waits are preserved by the transition relation, (b) if a position p is jammed then also any position to which the wafer at p may move next is jammed. We prove Proposition 5.5 and the technical lemmas only for our model of the EUV machine, but from the proofs it should be clear that these results can be generalized to a whole class of resource allocation problems.

Lemma 5.3 *Suppose that state s has circular wait in Q and $s \rightarrow s'$. Then state s' has circular wait in Q.*

PROOF. We consider three cases, corresponding to different types of transitions:

- If a process `entry_exit` takes a step, then this does not involve any position in Q: entry of a new wafer on positions in Q is not possible since all these positions are filled; also exit of a wafer in Q is not possible since for all positions in $q \in Q$ we have $\mathit{needs}^s(q) \neq \emptyset$. Since none of the variables in Q is modified, the fact that s has circular wait in Q implies that also state s' has circular wait in Q.

- Also if a process `move` takes a step then this does not involve any position in Q: entry of a new wafer on positions in Q is not possible since all these positions are filled; also exit of a wafer in Q is not possible since for all positions in $q \in Q$ we have $\mathit{needs}^s(q) \subseteq Q$. Hence the circular wait property is preserved by the transition.

- If a process `expose` takes a step, then this does not effect emptiness of positions, nor the value of the *needs* set. Hence the circular wait property is preserved by the transition, and also s' has circular wait in Q.

■

Lemma 5.4 *Suppose position p is jammed in state s and $p' \in \mathit{needs}^s(p)$. Then position p' is jammed in s.*

PROOF. By contradiction. Suppose p' is not jammed. Then there exists a path on which eventually p' is empty. If in this path, directly after p' becomes empty, we schedule a transition that empties p (this is possible since $p' \in \mathit{needs}^s(p)$), we obtain a path in which eventually p is empty. But we assumed no such path exists. Contradiction. ■

Proposition 5.5 *A state has a circular wait in some Q iff it is jammed.*

PROOF.

- $\Rightarrow$ Assume that state s has a circular wait in Q. Pick an element $q \in Q$ (this exists since s has circular wait in Q). By Lemma 5.3, any state s' reachable from s in zero or more steps has circular wait in Q. Hence, $s' \models (q \neq \mathtt{e})$. It follows that $s \models \mathbf{AG}(q \neq \mathtt{e})$. Therefore, state s is jammed.
- $\Leftarrow$ Assume that state s is jammed. Then there exists a position q such that q is jammed in s. Define q to be the least fixed-point $\mu Q(\{q\} \cup \mathit{needs}^s(Q))$. Then, by construction, $\mathit{needs}^s(q) \subseteq Q \neq \emptyset$. By Lemma 5.4, using an inductive argument, it follows that all positions in Q are jammed in s. This implies in particular that, for all $q \in Q$, $s(q) \neq \mathtt{e}$ and $\mathit{needs}^s(q) \neq \emptyset$ (the latter inequality follows because if $\mathit{needs}^s(q) = \emptyset$ this implies that q is a lock that is filled with an exposed wafer, so q can be emptied in a single transition, which contradicts the assumption that q is jammed). It now follows that state s has a circular wait in Q.

■

In the remainder of this paper, we will say that a state is *deadlocked* if it has circular wait, i.e., if it is jammed. The question that we need to answer is whether and how we can prevent the system of entering a deadlocked state. In Dijkstra's paper on the *banker's algorithm* [42], the first published deadlock avoidance algorithm, a state is defined to be *safe* if "all processes can be run to completion". In our case, the wafers are the processes and "a wafer is run to completion" if it exits the machine. Thus, Dijkstra's definition can be translated to CTL as follows.

Definition 5.6 (Safe states) *A state s is* safe *iff* $s \models \mathbf{EF}\left(\bigwedge_{p \in P}(p = \mathtt{e})\right)$.

Note that in general safe and not being deadlocked are different things. If a state s is not deadlocked then $s \models \bigwedge_{p \in P} \mathbf{EF}(p = \mathtt{e})$, i.e., each individual position can be emptied, but it need not be the case that all positions can be emptied simultaneously. If a state is deadlocked it is unsafe, but if it is unsafe it need not be deadlocked. However, in many cases and (according to SMV) in particular for our model of the EUV machine, the following property does hold for the initial state[2]:

$$\mathbf{AG}(\mathrm{safe} \iff (\mathbf{EG}\ \neg \mathrm{deadlock})). \tag{5.1}$$

[2]In fact, in the EUV machine a state is safe if and only if it has no deadlock. Thus, the RAS structure induced by the operation of the wafer scanner facilitates the application of the results presented in [76, 94]. It is, however, easy to come up with variations of the machine with states that are not safe and not deadlocked, for example a design in which the internal robots only have one arm. In such cases, in order to make formula (5.1) hold, we need to require weak fairness for all processes in the SMV model to exclude runs in which no progress is made due to infinite stuttering of some components.

This formula suggests a simple least restrictive DAP: just keep the system in a safe state. This policy can be realized for the EUV machine. Every non-initial safe state has at least one safe successor (different from itself), otherwise it would not be not possible to return to the initial state. In addition, we verified using SMV that all successors of the initial state are again safe.

5.3.4 A Least Restrictive DAP

In order to actually build a controller that always keeps the system in a safe state, it would clearly be very helpful to have a simple, yet exact characterization of the set of safe states. We see two ways to obtain such a characterization.

1. When checking whether the initial state is safe, SMV computes a *binary decision diagram* (BDD, see [33]) which provides a compact representation of the set of safe states.

2. The set of safe states can be manually characterized by a predicate expression P that is constructed by the following iterative procedure:

$$
\begin{array}{l}
P := \mathit{true} \\
\textbf{while}\ (s_{\text{init}} \not\models \mathbf{AG}(\text{safe} \iff P)) \\
\qquad P := P \wedge (\neg C)
\end{array}
$$

   where C is the characterization of the last state of the counter example that is generated by SMV.

The first approach enables a least restrictive DAP with linear time complexity, since checking whether a state is included in a BDD takes $\mathcal{O}(n)$ operations, where n is the number of booleans from which the BDD is composed (20 in case of the EUV machine). The size of the BDD, however, can in the worst case be exponential in the number of booleans. A second drawback is that it can be difficult to derive individual unsafe and/or deadlock situations from a BDD, which may be required during the design phase of the system. The second approach can quickly become practically infeasible since all unsafe states are explicitly enumerated. If it is carried out manually, however, then it might be possible to abstract from irrelevant state information and to visualize the various unsafe situations in the system. Of course, this requires some effort and creativity from the analyst. The second approach has been used to characterize the safe states of the EUV machine. With five iterations, we found four unsafe situations, depicted in Figure 5.3, which happen to characterize all deadlocks.

The predicate P that exactly characterizes the set of safe states is the negation of the situations shown in Figure 5.3, and which are described by predicates `d1`, `d2`, `d3` and `d4` in Figure 5.4.

Note that SMV can also be used to obtain a simple under-approximation of the set of safe states (when, e.g., the BDD is too large to use and the iterative process

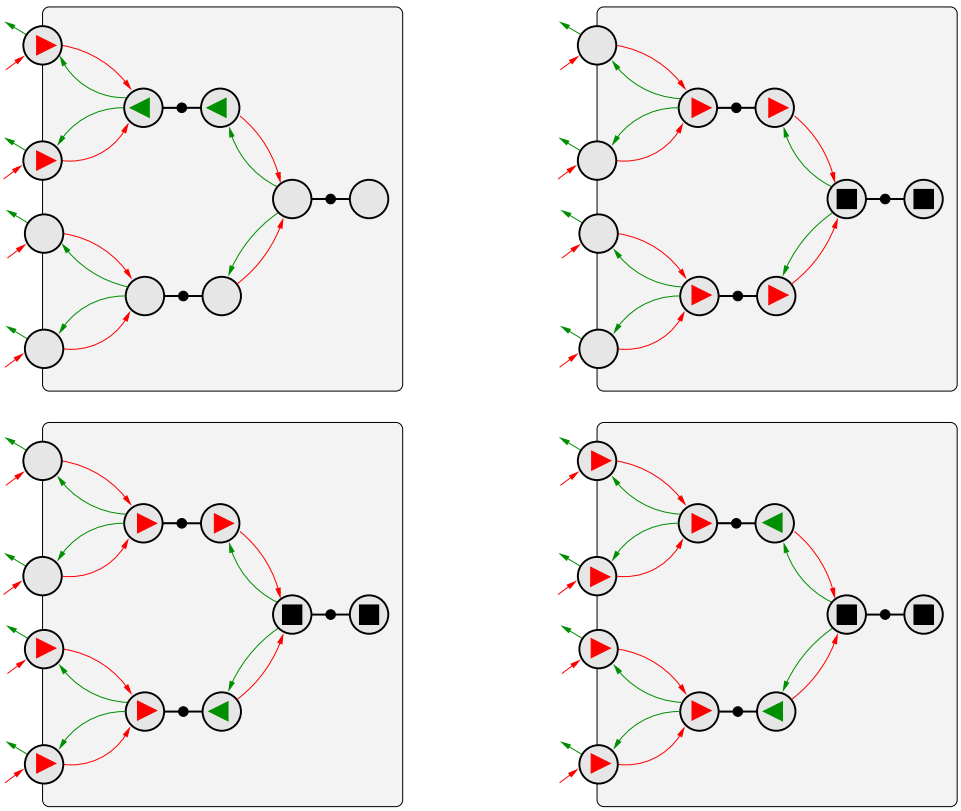

Figure 5.3: The four unsafe scenarios (modulo symmetry) in the EUV machine. A right-pointing arrow represents an unexposed wafer, a left-pointing arrow represents an exposed wafer, and a black square represents an unexposed or exposed wafer.

is too time consuming). If C is a candidate for a simple under-approximation, then this can be verified with the CTL property $\mathbf{AG}(C \Rightarrow \text{safe})$. Again, counter-examples can be used to correct C while retaining low complexity. Note, however, that it now becomes is necessary to ensure that the initial state is reachable from any state in C (this is true by definition for the set of all safe states).

5.4 Throughput Analysis

A first objective for a controller of the EUV machine is to avoid deadlocks. In the previous section, using our SMV model, we synthesized a least restrictive control policy that achieves this. A second key objective for a controller of the machine of course is to maximize throughput. Our SMV model is not sufficiently detailed to address this issue since, for instance, relevant information about the delays in the locks and the speed of the robots has not been included. Also, the SMV model

```
#define empty (l[0]=e & l[1]=e & l[2]=e & l[3]=e &
               rb[0][0]=e & rb[0][1]=e &
               rb[1][0]=e & rb[1][1]=e &
               c[0]=e & c[1]=e)

#define safe (EF(empty))

#define d1a (l[0]=r & l[1]=r &
             rb[0][0]=g & rb[0][1]=g)
#define d1b (l[2]=r & l[3]=r &
             rb[1][0]=g & rb[1][1]=g)
#define d1 (d1a | d1b)

#define d2 (~c[0]=e & ~c[1]=e & rb[0][0]=r &
            rb[0][1]=r & rb[1][0]=r & rb[1][1]=r)

#define d3a (~c[0]=e & ~c[1]=e & rb[0][0]=r &
    rb[0][1]=r & ((rb[1][0]=r & rb[1][1]=g) |
    (rb[1][0]=g & rb[1][1]=r)) & l[2]=r & l[3]=r)

#define d3b (~c[0]=e & ~c[1]=e & rb[1][0]=r &
    rb[1][1]=r & ((rb[0][0]=r & rb[0][1]=g) |
    (rb[0][0]=g & rb[0][1]=r)) & l[0]=r & l[1]=r)

#define d3 (d3a | d3b)

#define d4 (~c[0]=e & ~c[1]=e & l[0]=r &
    l[1]=r & l[2]=r & l[3]=r &
    ((rb[0][0]=r & rb[0][1]=g) |
    (rb[0][0]=g & rb[0][1]=r)) &
    ((rb[1][0]=r & rb[1][1]=g) |
    (rb[1][0]=g & rb[1][1]=r)))

safe_iff_p0: SPEC AG(safe <-> 1);
safe_iff_p1: SPEC AG(safe <-> ~(d1));
safe_iff_p2: SPEC AG(safe <-> ~(d1|d2));
safe_iff_p3: SPEC AG(safe <-> ~(d1|d2|d3));
safe_iff_p4: SPEC AG(safe <-> ~(d1|d2|d3|d4));
```

Figure 5.4: SMV characterization of the set of safe states.

abstracts from the delays due to turning of the internal robots, measuring of wafers, and swapping of the chucks. Therefore, in this section, we present a more refined *timed automata model* ([7, 8]), which contains sufficient information to address the throughput issue.

In order to define and analyze our model, we used the UPPAAL model checking tool. UPPAAL supports modeling of systems in terms of networks of timed

automata extended with blocking synchronization and bounded integer variables. Similarly to SMV, the semantics of a UPPAAL model is defined by a transition system. In addition to the discrete part, the states also contain a real-valued clock valuation. For these models, the UPPAAL model checker can decide a subset of *Timed Computation Tree Logic* (TCTL, see [6]). For a detailed account of UPPAAL we refer to [16] and to `http://www.uppaal.com`.

After presenting the UPPAAL model of the EUV machine in Section 5.4.1, we discuss the relationship between the UPPAAL and SMV models in Section 5.4.2. Then, in Section 5.4.3, we use UPPAAL to derive a schedule for the EUV machine that optimizes throughput.

5.4.1 UPPAAL Model

The UPPAAL model of the EUV machine contains the same state variables as the SMV model for the positions in the machine: arrays `l`, `rb` and c, which may take the same values `e`, `r` and `g` to indicate that a position is respectively empty, filled with an unexposed wafer, or with an exposed wafer. In addition, the UPPAAL model has a number of Boolean state variables to ensure "physical integrity":

- For each lock `id` there is a Boolean `lbt`[`id`] which is *true* iff either pressure in the lock is not atmospheric or in case a trackrobot is busy loading or unloading a wafer.
- Similarly, for each lock `id` there is a Boolean `lb`[`id`] which is *true* iff either the lock is not vacuum or in case an internal robot is busy loading or unloading a wafer.
- For each chuck `id` there is a Boolean `cb`[`id`] which is *true* iff either an internal robot is accessing chuck `c`[`id`] or an internal robot may not access chuck `c`[`id`] because it is not in location `measure` (i.e., it is busy with something).

The model consists of 12 automata, of which 11 model physical components of the machine: the trackrobot, the four locks, the four robotarms (two for each of the robots), and the two chucks. These automata move wafers around with certain delays and according to the material paths as specified in Section 5.2. An additional automaton, the *observer*, is used for throughput optimization.

Within the model a number of timing parameters are used. Figure 5.5 lists the values for these parameters that were provided by the designers of the machine.

Below, the individual timed automaton templates of the model are explained. Each template has a local clock x.

Figure 5.6 shows the trackrobot process. Initially, the trackrobot is ready to load a wafer to a lock. From its initial location, the trackrobot may move instantaneously to a location where it is ready to unload a wafer from a lock, but the reverse transition takes time *TR1*. When the trackrobot is ready to load, it may actually start loading a wafer to one of the four locks, provided the lock is empty

```
const H      1480;    const S        260;
const LOAD     25;    const UNLOAD    25;
const TR1      50;    const TR2       50;
const DEPRES  670;    const PRES     120;
const R2L_T    23;    const L2R_T     23;
const TURN     10;    const R2C_T     40;
const C2R_T    54;    const MEAS     140;
const EXPO    250;    const SWAP      10;
```

Figure 5.5: Timing parameters in the UPPAAL model.

and has atmospheric pressure. Similarly, when the trackrobot is ready to unload, it may start to unload a wafer from one of the locks, provided the lock contains a processed wafer and has atmospheric pressure (which is governed by the `lbt` variables). Upon finishing an unload operation the trackrobot synchronizes over the channel `unload` with the observer (which is explained below), and after *TR2* time units returns to its initial state.

Figure 5.7 shows the UPPAAL template for a lock. It has one parameter `id`, that provides the identity of this lock. Initially, a lock has atmospheric pressure. A lock may start depressurizing if the trackrobot is not busy with it. Similarly, if a lock is vacuum, it may start pressurizing if the internal robot is not busy with it.

There are two internal robots in the system, each equipped with two arms. Initially, one arm points at the chucks and the other arm points at the locks. An internal robot may turn, which interchanges the positions of the arms. Figures 5.8 shows the template for one type of robotarm, namely for the arms that initially point at the locks.

This template has four parameters: a constant *id* that identifies the internal robot to which the arm belongs, two constants *l0* and *l1* that identify the locks to which the robotarm has access, and a channel *turn*. When a robotarm is at the locks, then it can get a wafer from a lock (*L02R* and *L12R*), or it can put a wafer in a lock (*R2L0* and *R2L1*). Of course, it can only perform these actions if the lock is vacuum, and if the wafer flow is as specified in Section 5.2. Similarly, when a robotarm is at the chucks then it can load/unload a wafer to/from the chuck that is at the *measure* location. The *cb* variables are used to ensure that only one robotarm has access to the chuck at a time and that the chuck cannot execute a transition while the robotarm is loading/unloading a wafer. The template for the type of robotarm that initially points to the chucks is almost similar. It has another initial location, namely *at_chuck*, and it uses the "receiving" part of the channel (*turn?*) for proper synchronous turning of the two arms.

Figure 5.9 shows the UPPAAL processes for the chuck that initially is in the "measure" position. Like the robotarms, the chucks can simultaneously swap by synchronization over the channel *swap*. The *cb* variables are used by the chucks and the robotarms to prevent faulty behavior: (i) a robot can only access a chuck if it is in the measure position and not measuring (thus, the chuck must be in location

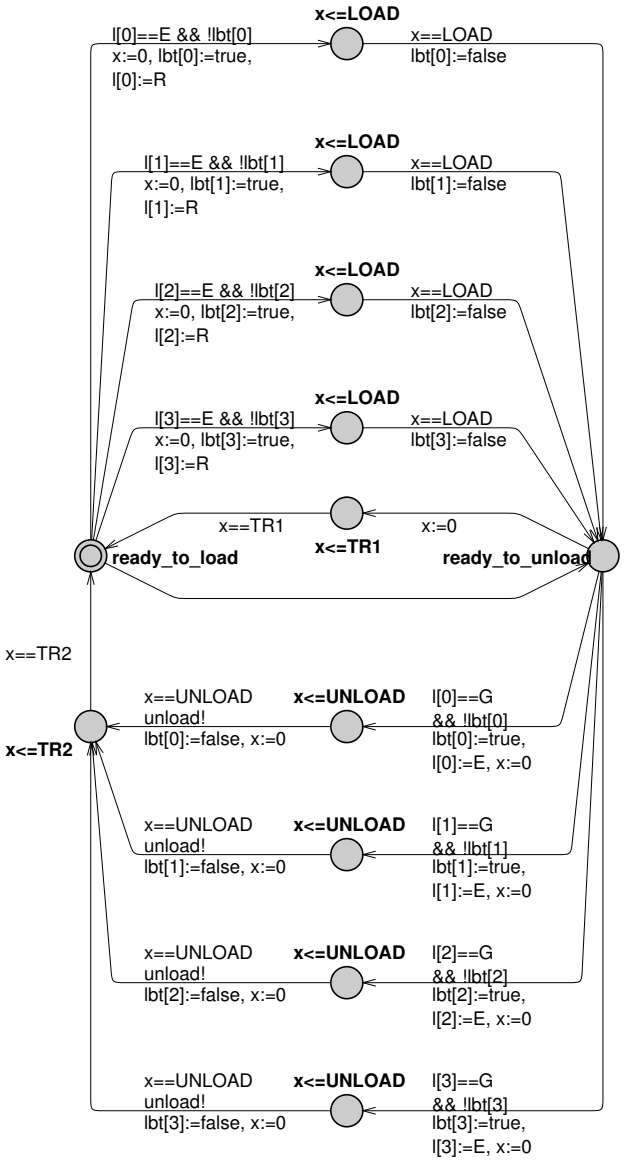

Figure 5.6: Process for the trackrobot.

measure), and (ii) when a robot is accessing a chuck, then the chuck may not perform any transitions. Each chuck has a local Boolean variable `m` which is *true* iff there is a measured wafer on the chuck; only a measured wafer can be exposed. The process for the chuck that initially is in the expose position is almost identical. It has *expose* as initial location, and it uses the "receiving" part of the channel (*swap?*) for proper synchronous swapping of the chucks.

Finally, Figure 5.10 shows the observer process which, as we will explain in more detail in Section 5.4.3, is used to ensure progress in the model. This pro-

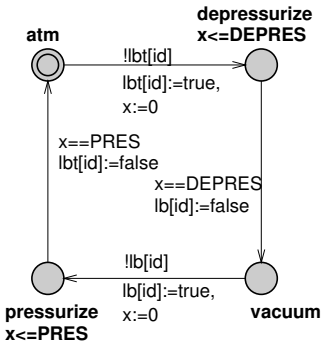

Figure 5.7: Template for a lock.

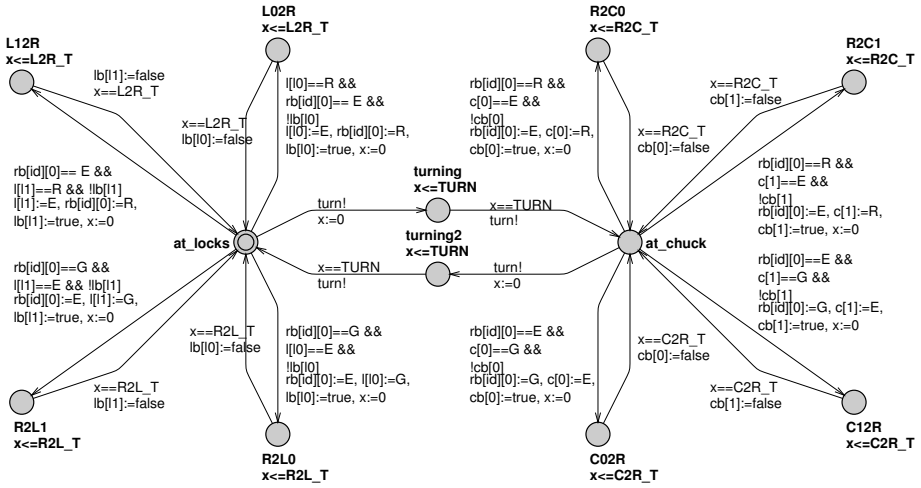

Figure 5.8: Template for a robotarm 0.

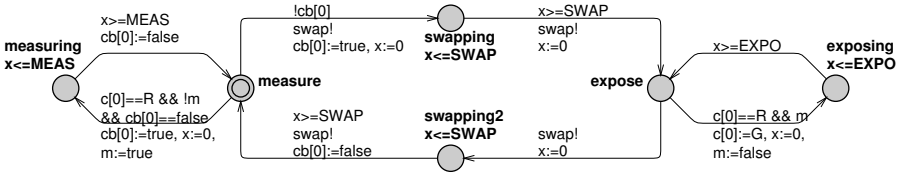

Figure 5.9: Process for chuck 0.

cess measures the time until the first wafer exits the system (this is signaled by the trackrobot over the channel *unload*) in location *L0*, and the time between two consecutive *unload* events in location *L1* using its local clock x.

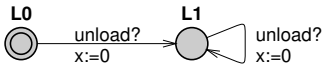

Figure 5.10: Process for the observer.

5.4.2 Bisimulation between SMV and UPPAAL models

Clearly, there is a relationship between the SMV model and the UPPAAL model. The SMV model is an abstraction from the UPPAAL model, which has the property that every transition in the UPPAAL model can be simulated in the SMV model, and vice versa. Formally, the relationship between the two models can be expressed as a *stuttering bisimulation* relation in the sense of [31]. Stuttering bisimulations are defined in terms of *Kripke structures*, an extension of transition systems in which to each state a set of atomic propositions is associated that hold in that state.

Definition 5.7 (Kripke Structures) *Let* $\mathbf{AP}$ *be a set of* atomic proposition symbols. *A Kripke structure is a structure* $(S, s_{init}, \rightarrow, l)$*, where* $(S, s_{init}, \rightarrow)$ *is a transition system and function* $l : S \rightarrow 2^{\mathbf{AP}}$ *associates to each state a set of atomic proposition symbols.*

In this paper, we let $\mathbf{AP}$ be the set of equations of the form $p = v$, where p is a position in the EUV machine and $v \in \{\mathtt{e}, \mathtt{r}, \mathtt{g}\}$. For the transition systems induced by the SMV and UPPAAL models, the labeling is obvious: we label a state s with $p = v$ iff this equation holds in s. For the SMV model the labeling function is injective: different states have different labels. For the UPPAAL model this is clearly not the case.

A stuttering bisimulation relates the states from two Kripke structures. Initial states are related, and related states are labeled with the same proposition symbols. If two states are related and from one state a transition is possible, then it should be possible to simulate this transition from the related state, after first doing zero or more *stuttering transitions*, i.e., transitions that do not change the labeling.

Definition 5.8 (Stuttering Bisimulation) *A* stuttering bisimulation *between two Kripke structures* $(S, s_{init}, \rightarrow, l)$ *and* $(S', s'_{init}, \rightarrow', l)$ *is a relation* $R \subseteq S \times S'$ *s.t.*

1. $(s_{init}, s'_{init}) \in R$,
2. *If* $(r, s) \in R$ *then* $l(r) = l(s)$,
3. *if* $(r, s) \in R$ *and* $r \rightarrow r'$ *then there exist, for some* $n \geq 0$, $s_0, s_1, \ldots, s_n$ *such that* $s_0 = s$ *and, for all* $i < n$, $s_i \rightarrow' s_{i+1}$, $(r, s_i) \in R$ *and* $(r', s_n) \in R$.
4. *if* $(r, s) \in R$ *and* $s \rightarrow s'$ *then there exist, for some* $n \geq 0$, $r_0, r_1, \ldots, r_n$ *such that* $r_0 = r$ *and, for all* $i < n$, $r_i \rightarrow r_{i+1}$, $(r_i, s) \in R$ *and* $(r_n, s') \in R$.

Proposition 5.9 *Consider the projection function π from states of the Kripke structure induced by the* UPPAAL *model to states of the Kripke structure induced by the* SMV *model. Function π only preserves the values of the arrays* `l`*,* `rb` *and* `c`*. Let R be the relation consisting of pairs $(s, \pi(s))$, for s a reachable state from the* UPPAAL *model. Then R is a stuttering bisimulation between the* UPPAAL *and* SMV *Kripke structures.*

PROOF. Function π maps the initial state of the UPPAAL model, in which the machine is completely empty, to the initial state of the SMV model.

By definition π, and hence R, preserves labeling of states.

Transfer property (3) follows by inspection of all the transitions in the UPPAAL model: each transition either does not affect the labeling, in which case it can be simulated by a stuttering transition in the SMV model, or it does affect the labeling but then a process in the SMV model is enabled that results in the same change of labels.

Proving transfer property (4) is somewhat more involved. We need a number of auxiliary invariants on the UPPAAL model. These include the integrity constraints mentioned at the beginning of Section 5.4.1 that restrict the values of the Booleans `lbt`[`id`], `lb`[`id`] and `cb`[`id`]. Also, we need some obvious invariants that relate the locations of connected robotarms, and the locations of the two chucks. The full set of invariants is listed in the file `EUV-invariants.q` which is available at `http://www.cs.ru.nl/ita/publications/papers/martijnh/`. The state space of the UPPAAL model is too big to establish these invariants directly. However, we were able to prove them automatically for an abstraction of the model in which we remove all clock variables, the arrays `l`, `rb` and `c`, as well as all references to these variables in transitions and locations. This is a valid abstraction in the sense that each invariant of the abstract model also holds for the original full UPPAAL model.

A key observation in the proof of transfer property (4) is that from any reachable state of the UPPAAL mode we can drive the system back to its initial state — except for the values of arrays `l`, `rb` and `c`, the values of the local clocks `x`, and the value of the `m` variables — by doing stuttering steps only. More specifically, let ρ be the mapping from states of the SMV model to states of the UPPAAL model such that, for any r, in state $\rho(r)$ the values of arrays `l`, `rb` and `c` are equal to the values in r, and all locations and Boolean variables have their initial value, except the two `m` variables, which are *true* iff the corresponding chuck contains an unprocessed wafer. Then we claim that, for any reachable state s of the UPPAAL model with $\pi(s) = r$, there exists a path with only stuttering steps from s to a state which, up to the values of local clocks, is equal to $\rho(r)$.[3] To see why this claim holds, first observe that each process in a non-quiescent location (a location with a nontrivial invariant) may evolve to a quiescent state by some stuttering transitions with guards that only refer to the local clock `x`, and (possibly) with synchronization

[3]Technically, the values of the clocks are irrelevant ("inactive") in the initial locations, and UPPAAL also abstracts from their value.

output labels (!) for which a corresponding input (?) is always enabled. After all processes have reached a quiescent state we can, one by one, drive each process back to its initial location:

1. The trackrobot only has two quiescent locations: ready to load and ready to unload. Via two successive internal transitions, we can drive the trackrobot from ready to unload to ready to load in time `TR1`.

2. Each lock has two quiescent locations corresponding to atmospheric pressure and vacuum. If a lock `id` is vacuum then, since the robotarms are in a quiescent state, due to the invariants, `!lb[id]`. Hence we can drive the lock to its initial location (atmospheric pressure) via two successive transitions in time `PRES`.

3. In order to bring the robotarms to their initial location, we may need to turn them around. The invariants for the robotarms imply that we can bring all arms in their initial location simultaneously.

4. For the chucks, we also have to ensure that `m` is *true* in case a chuck contains an unprocessed wafer. This can be achieved by driving the automaton through the measuring location, which may require swapping of the chucks. After the `m` variables have been set to the appropriate value, me may need to swap the chucks again. The invariants for the chucks imply that we can bring both chucks to their initial location.

If all processes are in their initial location, then the invariants imply that also the Boolean arrays `lb`, `lbt` and `cb` have their initial values. From this the claim follows.

Next, we claim for any state r from the SMV model that if s enables some transition, this can be simulated from $\rho(r)$, possibly after some stuttering steps. This follows from a routine case distinction. For instance, a transition moving a wafer from a lock `0` to an internal robot can be simulated by first depressurizing lock `0`, possibly turning the robotarm, and then moving the wafer to the robot via the transition to location `L02R`. We leave it to the reader to check the details of all the cases. ■

The significance of the above result stems from the fact that validity of CTL formulas without *nexttime* operator (i.e. all the formulas used in this paper) is preserved by stuttering bisimulation equivalence (see [31]). Thus, all the results on deadlock avoidance established using SMV in Section 5.3 carry over to the UPPAAL model. It is not possible to obtain these results directly using the UPPAAL tool since (a) UPPAAL does not support full CTL, and (b) the state space of the UPPAAL model is so big that it cannot be fully explored.

5.4.3 Finding an Optimal Schedule

As mentioned above, the process of Figure 5.10 observes unload events. It starts in location *L0* and upon the first unload event it resets its local clock x and enters location *L1*. In location *L1* the clock is reset whenever an unload event takes place. The observer is used to find an infinite schedule that takes at most H time units until the first unload event, and that has at most S time units between two unload events. Such a schedule is specified by the following TCTL property that can be checked by UPPAAL.

$$\mathbf{EG} \left(\begin{array}{c} (\mathit{observer.L0} \longrightarrow \mathit{observer.x} \leq H) \\ \wedge \\ (\mathit{observer.L1} \longrightarrow \mathit{observer.x} \leq S) \end{array} \right) \tag{5.2}$$

If this property is satisfied, then UPPAAL can return an example execution that consists of a path followed by a cycle. Such an execution thus gives an infinite control schedule for the wafer scanner with a *stationary* throughput of at least one wafer per S time units. Unfortunately, the size of the reachable state space prevents UPPAAL from finding such an execution directly. We therefore added heuristics to the model to prune the state space:

1. The DAP derived in the previous section has been used to avoid unsafe material configurations of the machine.
2. Some transitions are useless (or suboptimal) in certain states, e.g., an internal robot can always turn, but this is useless if it does not hold wafers. The state space has been reduced by adding guards that prevent such useless behavior.
3. The optimal behavior of the locks in the initial phase (the filling of the machine) differs from their optimal behavior in the stationary phase. Therefore a heuristic has been added to enforce this difference: a lock can pressurize when it contains either an exposed wafer, or it is empty and the machine is not yet filled with enough wafers to be in the stationary state.
4. Some transitions have been made *urgent* (greedy): they must be taken as soon as they are enabled. For instance, if the DAP allows loading a wafer to a lock, then this must be done immediately.

Note that using urgent transitions without the DAP may be an unwise idea, since this can result in many deadlocks with the effect that an execution satisfying Property 5.2 does not exist anymore in the model. Also note that at least the last three heuristics may remove good schedules.

A lower bound on the time until the first unload event, min_h, can easily be derived from the model. It is also easy to see that the minimal separation time between exposed wafers that appear at the chuck that is in the measure position (and can therefore be picked up by an internal robot) equals

$$\mathit{min}_s \;=\; \mathit{EXPO} + \mathit{SWAP},$$

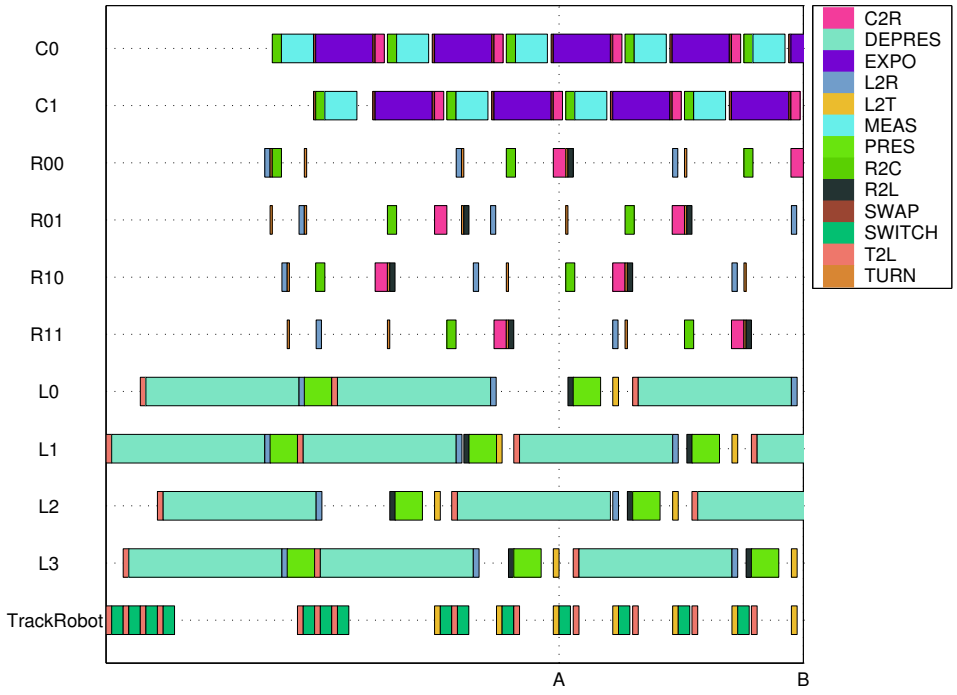

Figure 5.11: A schedule that optimizes the stationary throughput of the EUV machine. The cyclic part of the schedule consists of the interval between points A and B. The operation of the chucks (the EXPO and SWAP operations at C0 and C1) is critical in the cyclic part.

where the former is the time needed for the expose operation and the latter is the time needed for the chuck swap. Therefore, the theoretical maximal stationary throughput of the machine is at most one wafer per min_s time units. For the UPPAAL model with heuristics it is possible to find (in almost no time) an execution that satisfies Property 5.2 for a value of H that is 5% larger than min_h and for $S = min_s$. This execution was found by minimizing the values of H and S in formula 5.2 such that it can still be fulfilled. Figure 5.11 shows this schedule that thus optimizes the stationary throughput of the EUV machine.

It took only little effort to change the UPPAAL model in order to analyze two alternative machine designs with respect to throughput. In the first design alternative, the incoming wafers have been restricted to the upper two locks and the outgoing wafers to the lower two locks, in order to prevent deadlock a priori (see Section 5.2). Note that one lock has a wafer throughput of one wafer per

$$min_l = LOAD + PRES + DEPRES + L2R\_T$$

time units, where *LOAD* is the time needed by the track robot to place a wafer in the lock, *(DE)PRES* is the time needed to (de)pressurize a lock, and *L2R_T* is the time needed by an internal robot to grab a wafer from a lock. Thus, two locks

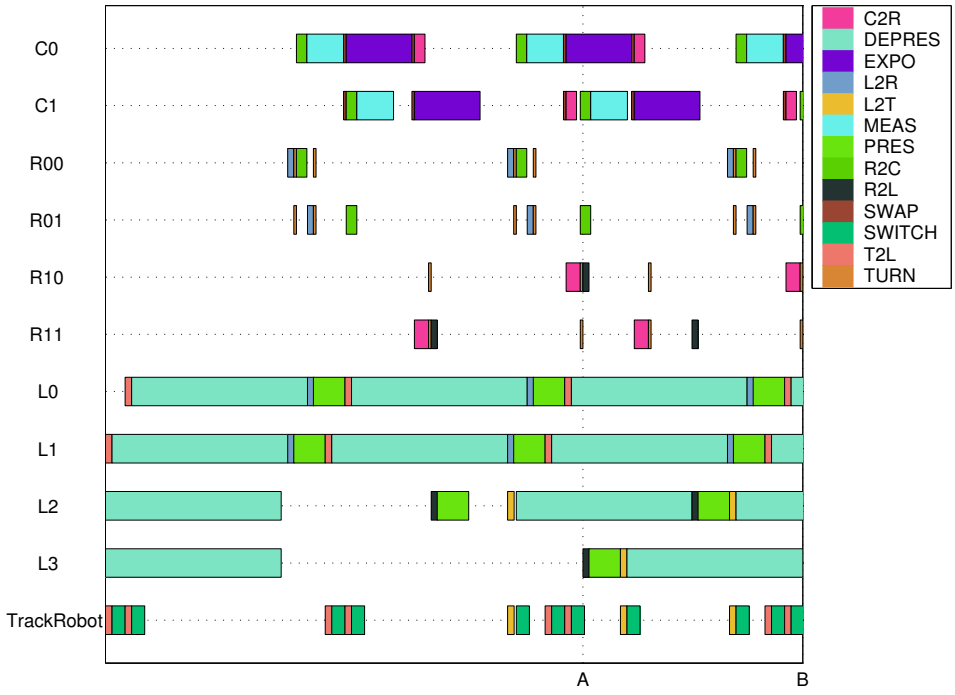

Figure 5.12: A schedule that optimizes the stationary throughput of the EUV machine in which unexposed wafers enter through the upper two locks (L0 and L1) and exposed wafers exit through the lower two locks (L2 and L3). The cyclic part of the schedule consists of the interval between points A and B. The operation of the locks is critical in the cyclic part.

have a throughput of at most one wafer per $\frac{1}{2}min_l$ time units. Since $\frac{1}{2}min_l > min_s$, a better upper bound on the throughput is 1 wafer per $\frac{1}{2}min_l$ time units. We are able to find a schedule for a value of H that is 11% larger than min_h and for $S = \frac{1}{2}min_l$. Therefore, this schedule optimizes the stationary throughput of this alternative machine lay-out. Concluding, the optimal stationary throughput is 61% smaller than the optimal stationary throughput of the original machine, and not the expose operation but the locks have become critical. This confirms our line of thought in Section 5.2. Figure 5.12 shows this alternative schedule.

The second design alternative consists of only two locks and one internal robot. Again, an upper bound on the throughput of this machine is 1 wafer per $\frac{1}{2}min_l$ time units. The throughput loss compared to the original machine thus is *at least* 61%. However, the best schedule we have been able to find with UPPAAL has a stationary throughput that is 83% worse than the optimal schedule for the original machine. Figure 5.13 shows this alternative schedule.

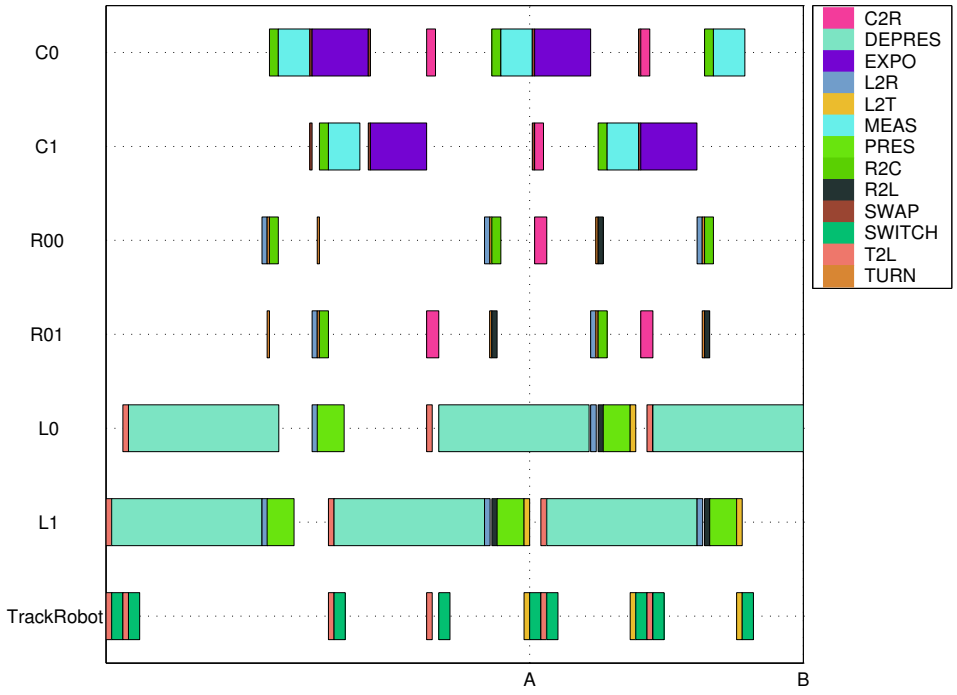

Figure 5.13: A schedule for the EUV machine with only two locks and one internal robot. The cyclic part of the schedule consists of the interval between points A and B.

5.5 Conclusions

The SMV model checker has successfully been used to characterize the set of safe states of the EUV machine. This characterization consists of a very short boolean expression over the places in the machine and is useful for the design of an actual controller since deadlock can easily be avoided by examining the possible successor states of the current state. Since the characterization is exact, the controller implements a least restrictive (optimal) deadlock avoidance policy.

Furthermore, we used the UPPAAL model checker to compute infinite schedules for the EUV machine that optimize stationary throughput. It took little effort to change the UPPAAL model in order to analyze two alternative machine designs. In theory, our approach can be applied to a broad class of resource allocation systems. As always when using model checking, the state space explosion is the main problem for scalability.

Altogether, in our view, the present work nicely illustrates the usefulness of model checking techniques to support the design process of applications that involve resource allocation and scheduling. Building models that are just abstract enough for addressing a specific question, often provides a good way to deal with the state space explosion problem.

A nice topic for future research would be to add probabilities to the picture.

The timing of the various robot and exposure operations in the EUV machine is known very precisely and exhibits minimal variability. So for these operations our deterministic model appears to be the right choice. However, the delay involved in the operation of the locks (pressurization and depressurization) is variable and for this a stochastic model makes sense. It would be interesting to carry out an analysis along the lines of [22] in which the quality of schedules that we computed using UPPAAL is assessed with respect to timeliness, utilization of resources, and sensitivity to different assumptions about the stochastic behavior of the EUV machine.

Acknowledgements. The authors thank Biniam Gebremichael for his useful suggestions concerning the SMV model, and the anonymous reviewers for their helpful comments on preliminary versions of the present paper.

Chapter 6

Production Scheduling by Reachability Analysis

GERD BEHRMANN

ED BRINKSMA

MARTIJN HENDRIKS

ANGELIKA MADER

Abstract. Schedule synthesis based on reachability analysis of timed automata has received a significant amount of attention during the last couple of years. The main strength of this approach is that the expressiveness of timed automata allows – unlike many classical approaches – the modeling of scheduling problems of very different kinds. Furthermore, the models are robust against changes in the parameter setting and against small changes in the problem specification. This paper presents a case study that was provided by AXXOM, an industrial partner of the AMETIST project. It consists of a scheduling problem for lacquer production, and is treated with the timed automata approach. A number of problems have to be addressed for the modeling task: the information transfer from the industrial partner, the derivation of timed automaton model for the case study, and the heuristics that have to be added in order to reduce the search space. We try to isolate the generic problems of modeling for model checking, and suggest solutions that are also applicable for other scheduling problems. Finally, model checking experiments are discussed.

6.1 Introduction

Scheduling theory is a well-established branch of operations research, and has produced a wealth of theory and techniques that can be used to solve many practical problems, such as real-time problems in operating systems, distributed systems, process control, etc. [92, 91]. Despite this success, alternative and complementary approaches to schedule synthesis based on reachability analysis of timed automata have been proposed recently [49, 3, 2]. The main motivation of this previous work is the observation that many scheduling problems can very naturally be modeled with timed automata. Furthermore, the expressiveness of timed automata renders the models robust against changes in parameter settings and changes in the problem specification. It has been shown that this approach is not necessarily inferior to other methods developed during the last three decades [3].

This chapter is a slightly modified version of [15] and reflects the most recent experimental data.

The case study presented in this paper is one of the four industrial case studies of the European IST project AMETIST, which focuses on the application of advanced formal methods for the modeling and analysis of complex distributed real-time systems with dynamic resource allocation as one of its special topics. The application of timed reachability analysis to this problem is one of the main subjects of the project. Technical material related to this case study, and different approaches to its solution can be retrieved from the AMETIST website [1].

The remainder of this paper is organized as follows. The principles of the derivation of schedules by reachability analysis are sketched in Section 6.2. Section 6.3 contains a description of the case study. Modeling issues and the use of heuristics are discussed in Section 6.4. The results of our model-checking experiments are collected and discussed in Sections 6.5 and 6.6. Finally, Section 6.7 evaluates the model checking approach to the case study and concludes the paper.

6.2 Scheduling with Timed Automata

The synthesis of schedules using timed automata can be seen as a special case of control synthesis [82], and was first introduced by [49], and by [3]. In general, a model class that provides the possibility to represent system events as well as timing information is suitable for real-time control synthesis. In this paper, the timed automata of Alur and Dill are used for modeling [8]. These timed automata extend the traditional model of finite automata with real-valued clock variables whose values increase with the rate of the progress of time. Clock variables can be reset to zero and they can be used in guards for discrete transitions as well as in guards for the elapse of time (this is used to ensure progress). In general, timed automata models have an infinite state space. The region automaton construction, however, shows that this infinite state space can be mapped to an automaton with a finite number of equivalence classes (regions) as states [8]. Finite-state model checking techniques can then be applied to the reduced, finite region automaton. A number of model checkers for timed automata is available, for instance, KRONOS [107] and UPPAAL [16].

Schedule synthesis using timed automata works as follows. First, a model of the unscheduled system is constructed. In our case, this model consists of the parallel composition of a number of timed automata. The non-determinism that is present in the parallel composition reflects the unresolved scheduling choices. Second, feasibility is formulated as a reachability property, for instance, “It is possible that the production is finished by Friday evening”. Third, the model checker exhaustively searches the reachable state space in order to check whether the property holds. If this is the case, then the model checker can provide a trace that proves the property. In our example, this is a trace from the initial state to a state in which the production is finished and it is not later than Friday evening. The information

[1] http://ametist.cs.utwente.nl/

contained in such a trace suffices to extract a feasible schedule, which is the final step of our approach.

The advantage of this approach is its (modeling) robustness against changes in the parameter settings and small changes in the problem specification. The disadvantage lies in the well-known state space explosion problem: the reachable state space is far too large to handle within a practical amount of time for many interesting cases. The approach that is used in this paper is to add heuristics and to use features of schedules that reduce the reachable state space to a size that can be searched more easily. We argue that these heuristics are quite general and applicable in many cases.

6.3 Description of the Case Study

The case study deals with a problem that is almost a job-shop problem [91], extended by parallel use of resources and additional timing restrictions between processing steps. Lacquers can be produced according to one of three *recipes*, for the lacquer types uni, metallic or bronce. A recipe specifies the processing steps, the resources needed for a processing step, the processing time, and timing constraints between processing steps. See Figure 6.1 for a graphical description of the recipes. There is a restricted number of resources available, such as mixing vessels, dose spinners, filling lines, etc. The problem is to schedule a number of lacquer *orders* Each order is specified by a lacquer type (i.e., recipe to use), release date, and due date.

As mentioned above, the problem has a lot in common with job-shop scheduling. The main differences are (i) there are additional timing restrictions between production steps (e.g., there must be at most 4 hours between the end of the first production step and the start of the second production step for uni lacquers), and (ii) an order must use resources in parallel (every lacquer needs a mixing vessel during its production in parallel with other resources).

There are two additional features of the case study that need explanation. First, an *availability* factor is associated with every resource. This factor models the fraction of the time that a resource is available due to the working hours of the personnel. E.g., if a resource is operated by personnel that works in two 8 hour shifts from Monday 6 am to Friday 10 pm (i.e., 16×5 hours per week), then the availability factor of that resource equals $\frac{80}{7\times24}$. This availability factor is used to take the working hours constraint into account by extending the processing times: if a processing step needs P minutes of processing time on a resource that has availability factor A, then the processing time is extended to $\frac{P}{A}$, for the example above that would be $P \times \frac{7\times24}{80}$. The use of the availability factor is intended for approximation in long-term scheduling, i.e., the question how many orders can be done within the next half year. For daily scheduling the actual working hours have to be modeled. Second, a *performance* factor is associated with every resource. This factor models the fraction of the time that a resource is unavailable due to

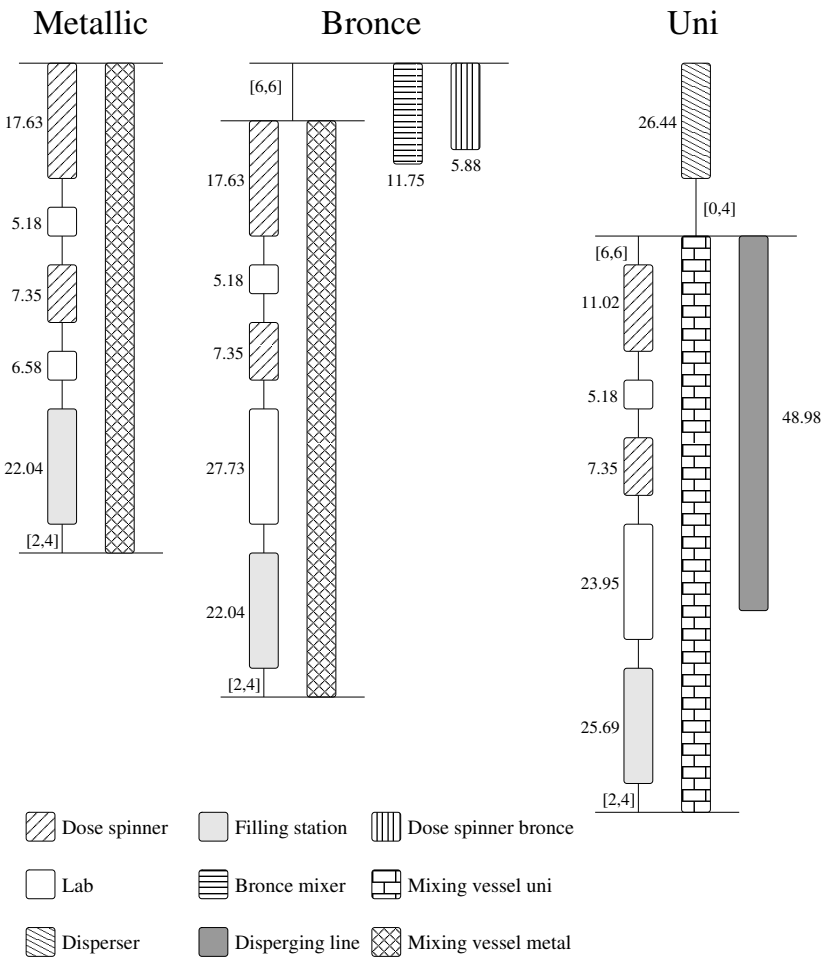

Figure 6.1: An alternative graphical representation for the three recipes.

break-downs or maintenance. The performance factor is used in the same way as the availability factor to extend the processing times. Note that both availability factors and performance factors are given by AXXOM, as well as the mechanism to extend processing times. Section 6.6 discusses these performance factors in more detail. Seven slightly different versions of the case study have been examined:

1. The performance and availability factors are used to extend the processing times of the 29 orders. Furthermore, the processing times do not depend on the size of an order (i.e., producing 15000 kg takes as much time as producing 5000 kg).

2. Same as (1), but with 73 orders instead of 29.

3. Same as (2), except that the orders are "vertically multiplied" n times. I.e., this essentially means that model 2 is scheduled n times (for a given n).

4. Same as (2), except that the orders are "horizontally multiplied" n times with proper replication of the resources.

5. The processing times are a function of the size of an order, storage costs are added for the final product, and delay costs are added for all processing steps to quantify the quality of schedules. Furthermore, the model of some resources has been made more exact (for instance to model setup times between processing steps). The performance and availability factors are used to extend processing times.

6. Same as (5), except that an exact model of the working hours constraint is used. The performance factor still is used to extend processing times.

7. Same as (6), except that this model also includes storage costs for the intermediate products.

6.4 Modeling with Timed Automata

This section explains how the problem has been modeled with timed automata. We intend to make the models that have been used in this paper available on the AMETIST website as soon as possible.

6.4.1 Information Transfer from Industry

A substantial amount of the time spent on the case study went into the modeling activities. The most difficult part here was the information transfer from AXXOM to the academic partners. In the first place, there was a language problem regarding the domain specific interpretation of terminology. For this purpose we compiled an initial dictionary in which relevant terms used are explained in natural language. This dictionary served as an agreement with AXXOM on the main, basic facts. Additionally, there was a documentation problem, regarding the (implicit) knowledge that always exists beyond any written specification. This problem remained present even after agreeing on the dictionary. This suggests that, beyond a dictionary, additional validation of the basic facts would be desirable.

Another difficulty was caused by the format that AXXOM used for the recipes, which was neither standard, nor intuitive. A better (from the computer science perspective, at least) representation had to be devised, which resulted in the description shown in Figure 6.1. This new notation also helped to detect other gaps in the case description.

Finally, AXXOM is not working on lacquer production, but develops tools for value chain management. The case description they provided is to some extent the description of their own model of the original case. Making our timed automata models we faced the problem that we were remodeling another model, tailored for another tool, rather than the original case. One example in point are the occurrence of very high delay costs. In the AXXOM tool these are needed to simulate hard

deadlines by using (soft) due dates. With timed automata, hard deadlines can be modeled directly.

6.4.2 Timed Automaton Models

The lacquer production case is very similar to the job shop scheduling problem, involving just a few additional timing constraints, and the basic modeling by timed automata roughly follows [3]. Each processing step can be mapped to a sequence of three locations in a timed automaton (fragment), see Figure 6.2, where the transition between the first two locations claims the resource, the second location represents the processing period, and the transition to the last location frees the resource.

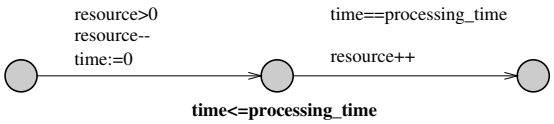

Figure 6.2: A single processing step modeled as timed automaton fragment.

The sequential and interleaved composition of the automaton fragments follows the descriptions and timing restrictions in the recipes. For each recipe there is a timed automaton (template) with free parameters for release date and due date. Five resources are modeled as counters, and the remaining resources are modeled as small automata (since these resources need their own clock). The parallel composition of the instantiated automata and the resource automata forms the system model.

When looking for feasible schedules we checked the reachability property "all orders (automata representing an order) reach their final state", where a guard in the model only allowed to enter the final state if the due date has not passed already.

6.4.3 Modeling Heuristics

The heuristics we used are more or less standard in operations research, and are thus not specific for this case study. For instance, the "non-laziness" heuristic as explained below is the same as considering only "active" schedules [91]. The modeling of these heuristics can be seen as standard patterns that can be re-used for similar cases. Each heuristic reduces the search space. We distinguish two kinds of heuristics. First, there are "nice" heuristics, for which we know that for each good schedule that was pruned away there is a schedule in the remaining search space that is at least as good. Second, there are "cut-and-pray" heuristics for which there is no such guarantee (i.e., the optimal schedule may be lost). Below we describe each of the heuristics we used, and show our modeling into the timed-automaton framework.

Non-overtaking

This heuristic is applied within each group of orders following the same recipe. It says, that an order started earlier also will get critical resources earlier than an order started later. This heuristics makes sense if for every two orders it holds that if the start time of the first order is smaller than the start time of the second order, then the end time of the first order is smaller than the end time of the second order. It is easy to see that for two orders following the same recipe, a non-overtaking schedule can be constructed from a schedule with overtaking. This can be done if at each moment when a resource is assigned to the later order (overtaking moment), we give it instead to the earlier order. This obviously is also a "nice" heuristic, if the processing times have the same length.

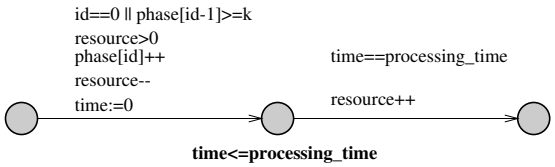

Figure 6.3: A timed automaton fragment for taking a resource with non-overtaking.

Technically, we divided an order in phases that roughly reflect the processing steps and that are numbered from 1 upwards. Non-overtaking has been realized by counters keeping track of the phase in which the order is. When a phase (processing step) is entered and a resource is taken, the counter is incremented. A restriction for entering a phase is that the previous order already has entered the phase before. For this purpose we have indexed counters *phase[id]*, one for each order having the number *id*. Note, that the sequence of identification numbers *id* reflects the sequence of release dates (and due dates, because the maximal production periods are the same). The order with identification number 0 is the first and may enter new phases without restriction. In Figure 6.3 we extended the basic timed-automaton fragment of Figure 6.2 by the counter construction. The constant k represents the k-th phase of the order. Note also that we have an indexed counter for each set of orders following one of the three recipes (thus, an order for metallic lacquer may overtake an order for a uni lacquer).

Non-laziness

In operations research non-lazy schedules are called active. The following behavior is excluded: a process needs a resource that is available, but it does not take the resource. Instead, the resource *remains unused*: no other process takes it. Then, after a period of waiting the process decides to take the resource. (And we regard this waiting time as wasted, which is only true if there are no timing requirements for starting moments of subsequent processes.) This is a "nice" heuristic.

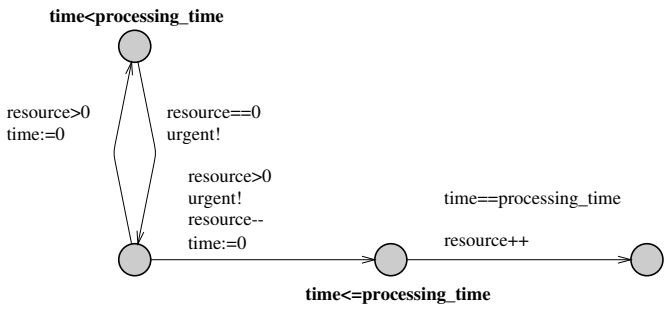

Figure 6.4: A timed automaton fragment for taking a resource with nonlaziness.

In [28] it is shown that non-laziness can be expressed directly in timed automata. Technically, we extended the basic timed automaton fragment of Figure 6.2 by an extra location that is entered if the resource is available, but not taken as depicted in Figure 6.4. The new location can only be left, if there is another order taking the resource. If for *processing_time* the resource has not been taken, a deadlock is caused, which has the effect of back-tracking and searching for other solutions. The intuition is, that if the resource has not been taken for *processing_time* the actual order could have taken it without being in the way for another order. Note that we use urgent communication on channel *urgent* so that some transitions are taken immediately if their guards become true, or pre-empted immediately by another enabled transition, if it exists. To make this work an automaton continuously offering synchronization on the *urgent* channel by a simple selfloop in its only location is part of the model. In the initial location of the automaton of Figure 6.4, therefore, when a resource becomes available it is either taken immediately or the idling state is reached immediately.

Greediness

This is a "cut-and-pray" heuristic. If there is a process step that needs a resource that is available, then the process step claims this resource immediately. By this it excludes possibly better schedules where some other (more important, because closer to deadline) process would claim the same resource shortly later. Note that greediness is stronger than non-laziness, i.e., every greedy schedule is non-lazy.

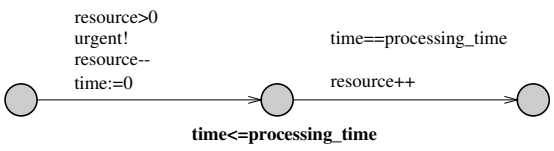

Figure 6.5: A timed automaton fragment for taking a resource with greediness.

Modeling greediness in a timed automaton is easy: the requirement is that a

resource has to be taken as soon as it is available. The communication via an urgent channel forces to take the transition as soon as the guard *resource*$>$ 0, becomes true, see Figure 6.5.

Reducing the Number of Active Orders

When not restricting the number of active orders (i.e., the orders that are processed at a certain moment), it often happens that many processes fight for the same resources, and block other resources while they wait. In our example the dose spinners (2 instances of these resources are present) have to be used by each process twice. Restricting the overall number of active orders avoids analysis of behavior that is likely to be ineffective. This heuristic is very powerful, but belongs to the "cut-and-pray" type. Technically, we realized this heuristics by a global variable that is increased when an order starts and decreased when an order is finished. A start condition for an order is that the counter has not reached a predefined upper bound.

Increasing the Earliest Starting Time of Orders

This is a very simple heuristic that we use in the models that include costs. Ideally, an order is finished right on its deadline: it then has neither storage nor delay costs. Thus, when many orders are finished too early, their starting times may be increased to reduce the costs.

6.4.4 Modeling the Extended Case Study

Some constraints have been approximated in the basic case study (i.e., in models 1 – 4) to simplify the problem. In this section we discuss the extension of the model to cope with the full constraints. We begin with an informal explanation of these constraints.

First, there are setup times and costs. The filling lines must be cleaned between two consecutive orders if those orders are not of the same type. Thus, additional cleaning time (5 – 20 hours) is needed and there is a certain cost involved with cleaning. Modeling this constraint poses no problems. Instead of modeling the filling lines by an integer variable, they are now each modeled by an automaton that keeps track of the type of the order that has last been processed by it.

Second, there are delay and storage costs. The happiness of a customer decreases linearly with the lateness of his order. Thus, each order has a delay cost, which is a "penalty" measured in euros per minute. Similarly, if an order is finished too early, then it has to be stored and this also costs a certain amount of euros per minute. In the initial problem, the costs are approximated by requiring that every order must be finished before its deadline. A more refined cost model enables us to prefer an order that is five minutes late to an order that is weeks early. UPPAAL

CORA[2] is a version of UPPAAL for cost optimal reachability analysis in *linearly priced timed automata*. UPPAAL CORA enables us to model delay and storage costs in a natural way [75]. It allows the representation of costs as affine functions of the clock variables. For instance, Figure 6.6 depicts how delay costs are modeled. Every order has a delay cost factor (*dcf*) that gives the cost per time unit when the order is too late. Furthermore, every job has a function *isLate()* that returns 0 when the due date of the order has not yet passed, and 1 otherwise. Every location of the order automaton in which the order is not yet finished then is equipped with a specification of the time-derivative of the cost: *cost'==isLate()*dcf*. A similar strategy is followed for modeling the storage costs. Every location of the order automaton in which the order is finished is equipped with *cost'==!isLate()*scf*, where *scf* is the storage cost factor.

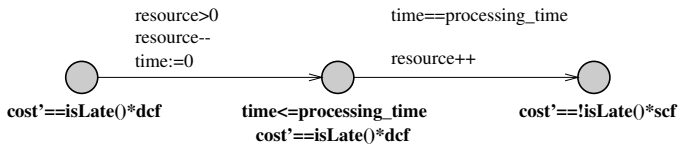

Figure 6.6: A timed automaton fragment for costs.

Third, there is the working hours constraint. The lacquer production is overseen by personnel that works in two or three shifts, depending on the machine they operate. Furthermore, the production is interrupted in weekends. Note that this constrained is approximated in the initial problem by the *availability factor* of machines. Another complicating factor is that some production steps may only be interrupted for 12 hours. Modeling the working hours constraint proved to be quite involved. A separate automaton was added that computes the *effective* processing time e, given the current time and the net processing time c. For instance, if the current time and c are such that the processing must be interrupted, then $e = c + B$, where B equals the length of the interruption. The additional automaton is rather big and laborious to produce, but quite logical in structure.

6.5 Model Checking Experiments

In Table 6.1 we collected models and model checking experiments for the feasibility analysis and schedule synthesis of the basic case study (i.e., models 1–4). The results were obtained using UPPAAL 3.5.6 on 2.6 GHz Intel-P4-Xeon processor and 2.5 GB of memory running Linux kernel 2.6.8.

Initial experiments revealed scalability problems, partly caused by the large number of clocks contained in the models. The heuristic limit of the number of active orders also provides a limit on the number of clocks needed (one per active order instead of one per order), and the non-overtaking heuristic provides an easy

[2] `http://www.cs.aau.dk/~behrmann/cora/`

Table 6.1: Experiments for the generation of delay-free schedules. Abbreviations for the heuristics are: *g* – greedy, *nl* – non-lazy, *no* – non-overtaking. Each experiment was repeated for a model without both availability and performance factors (column *no av, no pf*), a model with only availability factors (column *av, no pf*), and a model with both availability and performance factors (column *av, pf*). For each experiment the run-time in seconds is provided. All measurements were done using depth-first search. A run was terminated after 10 minutes, or when memory consumption reached 2 GB (indicated by a "-" in the table).

model #	# orders	heuristics	max. active	no av, no pf	av, no pf	av, pf
1	29	-	-	78.0	-	-
1	29	nl	-	0.3	1.5	3.5
1	29	nl no	-	0.3	2.5	2.6
1	29	g	-	0.2	2.8	-
1	29	g no	-	0.2	2.9	-
2	73	-	-	-	-	-
2	73	nl	-	-	-	-
2	73	nl no	-	24.6	-	-
2	73	nl no	3	0.4	-	-
2	73	nl no	4	1.1	0.4	0.3
2	73	nl no	5	4.4	0.6	0.9
2	73	g	-	15.0	-	-
2	73	g no	-	9.2	43.3	30.5
2	73	g no	3	0.3	165.1	-
2	73	g no	4	0.4	0.3	0.3
2	73	g no	5	2.2	0.4	6.8
3	146	g no	4	1.3	0.9	0.8
3	584	g no	4	37.7	21.2	18.6
3	1168	g no	4	166.4	92.1	80.7
3	2044	g no	4	565.4	311.6	271.9
4	146	g no	5	9.7	-	-
4	146	g no	6	167.8	-	-
4	146	g no	7	-	21.0	-
4	146	g no	8	-	18.2	12.6
4	219	g no	8	-	-	-
4	219	g no	9	-	-	347.9
4	219	g no	10	-	-	-

way of uniquely assigning shared clocks to orders, since it fixes the starting order of orders of a particular type. This change reduced the number of clocks to $3 \cdot A + 3$, where A is the maximum number of active orders.

The results show that, even for the case of the 29 orders, the use of heuristics is essential. The non-overtaking heuristic does not make much difference for a case without availability factors, whereas in the case with availability factors the performance gets worse. This might be caused by additional deadlocks in the state space that would otherwise have led to a schedule. For 73 orders and more, however, non-overtaking decreases the computation time considerably. Limiting the number of active orders increases the speed by several orders of magnitude (partly due to the possible reuse of clocks).

Experiments have been performed also for the extended version of the case study (i.e., models 5–7) using UPPAAL CORA, see Table 6.2. Although the constraint of the explicit working hours (used in the models 6 and 7) makes the model much more complex, the results show that their schedules have much lower cost than the schedules for model 5. A possible explanation is that the availability factors distribute the availability uniformly over time. In reality, however, a resource is available during the weekdays and completely unavailable in the weekends. Therefore, the availability factors give in some cases a large over-approximation of the processing times, with the result that scheduling becomes much harder.

Table 6.2: Table of experiments for the versions including costs with performance factors (pf) for all models, availability factors (av) in model 5, and explicit working hours (ex) in the models 6 and 7. A random-best-depth-first search of 10 minutes was used for the experiments with models 5 and 6. The experiments with model 7 (which used the same search order) had to be limited to 10 seconds due to problems with running UPPAAL CORA. The number of successful runs (termination rate), the lowest cost of any run and the average cost of all runs is shown.

model #	**# orders**	**heuristics**	**pf / av / ex**	**max. active**	**search time**	**termination rate**	**min. costs**	**average costs**
5	29	g	yes/yes/no	5	10 min.	10/10	$11 \cdot 10^6$	$13 \cdot 10^6$
6	29	-	yes/no/yes	5	10 min.	10/10	$2.1 \cdot 1^0 6$	$2.7 \cdot 10^6$
7	29	-	yes/no/yes	5	10 s.	3/10	$80 \cdot 10^6$	$81 \cdot 10^6$

The meaning of the costs is as follows: in model 6, a schedule with a cost of approximately 2 million (the best schedule of the 10 runs) is a schedule in which 2 orders are a bit late (1 day and 45 minutes respectively) and the others are much

too early (since intermediate products do not incur storage costs). In model 7, this effect is countered by storage costs for intermediate products, which makes the schedules more expensive. The investigation of these effects is still ongoing work. It should be noted that the non-laziness heuristic is not applicable to the extended case, since storage costs make it profitable to be lazy. Also the non-overtaking heuristic is rendered a "cut-and-pray" by the addition of costs. The greediness heuristic is not applicable either, but this is due to a limitation in UPPAAL. Still, we derived schedules that are competitive with those provided by AXXOM.

6.6 Stochastic Analysis

As explained earlier, so-called performance factors are used to indicate the percentage of time that a resource is unavailable due to maintenance and break-downs. The way in which AXXOM deals with this information is that the processing time on each resource is extended by the corresponding factor. E.g., if a machine only is available half of the time, the processing time for each processing step using this resource is doubled. Schedules are derived assuming that the process durations are extended in this way. This raised the question on the interpretation of the schedules derived with the extended processing times. Stochastic analysis [22] showed that the schedules derived in this way have less chance to reach the due dates than schedules without extended times. The interpretation roughly is as follows: if we reserve time for break-down when a resource is actually available, this time is simply wasted. Later, when the resource really breaks down, there will be too little time left to reach the due date. A conclusion is that extending processing times may give a useful indication how many orders can probably be done within a long time interval, say a few months, but it does not help for daily fine-tuned scheduling.

6.7 Evaluation and Conclusion

We have shown that feasible schedules for a lacquer production case can be derived doing real-time reachability analysis with the timed automata model checker UPPAAL. We could treat instances ranging from 29 to 2044 orders (all within 10 minutes, given the right heuristics). To deal with the full set of constraints of the original problem we had to introduce costs into the model, viz., setup costs for filling stations, storage costs for orders that are produced too early, and delay costs for orders that are too late. This transformed the problem into a cost-optimization problem, which was treated using UPPAAL CORA, a cost-optimizing version of UPPAAL. A further extension of the model was needed to deal with the so-called working hours constraints, which increased the size and complexity of the model significantly. Yet, also for this case competitive schedules could be derived using UPPAAL CORA.

On the one hand, it is clear that this application of model checking techniques to this kind of production scheduling problems is not (yet) push-button technol-

ogy: to obtain results our models had to be constructed with care, and the right heuristics had to be identified. On the other hand, it is reasonable to assume that many production scheduling problems have similar ingredients and that modeling techniques and patterns for typical plant processes and heuristics can be reused. Further experiments have to be carried out to identify a useful core collection of such modeling patterns.

The use of performance and availability factors leads to questions of interpretation. Extending the processing times by these factors can be used to analyze how many orders should be feasible on a longer time scale. The stochastic analysis in [22], however, has shown that using performance and availability factors to obtain concrete schedules increases the probability to miss deadlines. The use of these factors thus makes models inherently approximative, and it does not seem very useful to include finer information about penalties (such as setup and cleaning costs) into the model, as is the case now. It is unclear what modeling assumptions are best suitable for the derivation of concrete short-term schedules, where storage costs have to be minimized and delay costs to be avoided. An idea that we want to explore is that of using a form of schedule refinement taking rough long-term schedules as a basis for obtaining precise schedules for concrete short-term. A transformation approach to scheduling, although in a different context, was successfully used in another case study of the AMETIST project, viz., the CYBERNETIX case [81]. Another idea that will be investigated is that of searching for schedules in reverse time, starting from the due dates of orders; valid schedules obtained this way avoid storage and delay costs by construction.

The case study also raised a number of pragmatic questions concerning the modeling process. It turned out to be nontrivial to obtain all relevant information from AXXOM. In spite of our efforts to create a dictionary and better graphical representations, the models had to be changed substantially in an advanced stage of the project, as initially provided requirements turned out to be over-specified. The experience suggests that beyond a dictionary, there should have been some joint activity to certify the informal explanations. A related aspect is that the problem description of AXXOM was strongly influenced by the capabilities of their own planning tool. This implies that in some places we may have been remodeling the AXXOM model, rather than modeling the original problem.

Summarizing, we can say that our experience with the AXXOM case study shows that the application of model checking techniques for production scheduling is very promising. Still, considerable further work on modeling methods, reusability of modeling patterns, identification and evaluation of heuristics – all in the context of case studies of greater orders of magnitude – is needed to develop it into a readily applicable standard technique for schedule synthesis.

Acknowledgements. We would like to acknowledge the essential role played by Dagmar Ludewig and Sonja Loeschmann of AXXOM, who were always willing to provide the answers to our many questions concerning this case study.

Chapter 7

Model Checking the Time to Reach Agreement

MARTIJN HENDRIKS

Abstract. The timed automaton framework of Alur and Dill is a natural choice for the specification of partially synchronous distributed systems (systems which have only partial information about timing, e.g., only an upper bound on the message delay). The past has shown that verification of these systems by model checking usually is very difficult. The present paper demonstrates that an agreement algorithm of Attiya et al, which falls into a – for model checkers – particularly problematic subclass of partially synchronous distributed systems, can easily be modeled with the UPPAAL model checking tool and that it is possible to analyze some interesting and non-trivial instances with reasonable computational resources. Although existing techniques are used, this still is an interesting case study since it shows that the class of partially synchronous distributed systems now slowly comes within reach of mechanical verification techniques. Furthermore, the agreement algorithm has not been formally verified before to the author's knowledge.

7.1 Introduction

Distributed systems are in general hard to understand and to reason about due to their complexity and inherent non-determinism. That is why formal models play an important role in the design of these systems: one can specify the system and its properties in an unambiguous and precise way, and it enables a formal correctness proof. The I/O-automata of Lynch and Tuttle provide a general formal modeling framework for distributed systems [80, 79, 78]. Although the models and proofs in this framework can be very general (e.g., parameterized by the number of processes or the network topology), the proofs require – as usual – a lot of human effort.

Model checking provides a more automated, albeit less general way of proving the correctness of systems [37]. The approach requires the construction of a model of the system and the specification of its correctness properties. A model checker then automatically computes whether the model satisfies the properties or not. The power of model checkers is that they are relatively easy to use compared to manual verification techniques or theorem provers, but they also have some clear drawbacks. In general only *instances* of the system can be verified (i.e., the algorithm can be verified for 3 processes, but not for n processes). Furthermore, model checking suffers from the state space explosion problem: the number of states

This chapter is an almost literal copy of [56].

grows exponentially in the number of system components. This often renders the verification of realistic systems impossible.

A class of distributed systems for which model checking has yielded no apparent successes is the subclass of *partially synchronous systems* in which (i) message delay is bounded by some constant, and (ii) many messages can be in transit simultaneously. In the partially synchronous model, system components have some, possibly incomplete, information about timing. For instance, only an upper bound on the message delay may be known. It lies between the extremes of the synchronous model (the processes take steps simultaneously) on one end and the asynchronous model (the processes take steps in an arbitrary order and at arbitrary relative speeds) on the other end [78]. The timed automata framework of Alur and Dill [8] is a natural choice for the specification of partially synchronous systems (as is the Timed I/O-automaton framework [73], which, however, does not support model checking). Verification of the above mentioned subclass of "difficult" partially synchronous systems by model checking, however, is often very difficult since every message needs its own clock to model the bounds on message delivery time. This is disastrous since the state space of a timed automaton grows exponentially in the number of clocks. Moreover, if messages may get lost or message delivery is unordered, then on top of that also the discrete part of the model explodes rapidly.

Many realistic algorithms and protocols fall into the class of "difficult" partially synchronous systems. Examples include the sliding window protocol for the reliable transmission of data over unreliable channels [99, 35], a protocol to monitor the presence of network nodes [20, 72, 21], and the ZeroConf protocol whose purpose is to dynamically configure IPv4 link-local addresses [34, 51]. Furthermore, the agreement algorithm described in [10] (see also Chapter 25 of [78]) also is a partially synchronous system that is difficult from the perspective of model checking. The analysis of this algorithm with the UPPAAL model checker is the subject of the present paper. The main contribution consists of the formal verification of some non-trivial instances of the algorithm, which has not been done before to the author's knowledge. Although standard modeling and verification techniques are used, the case study still is interesting since it shows the current power of the UPPAAL tool and increases the existing body of case-study experience. Independently of the present work, Leslie Lamport has also analyzed a distributed algorithm that falls into the class of difficult partially synchronous systems as defined above [74].

The remainder of this paper is structured as follows. The timed automaton framework and the UPPAAL model checker are very briefly introduced in Section 7.2. Section 7.3 then presents an informal description of the distributed algorithm of [10], which consists of two parts: a timeout task and a main task. Section 7.4 describes the UPPAAL model that is used to verify the timeout task. A model for the parallel composition of the timeout task and the main task is proposed in Section 7.5. Two properties of the timeout task that have been verified in Section 7.4 are used to reduce the complexity of this latter model. Finally, Section 7.6 discusses the present work. The UPPAAL models from this paper

are available at `http://www.cs.ru.nl/ita/publications/papers/martijnh/`. Note that the UPPAAL development version 3.5.7 has been used.

7.2 Timed Automata

This section provides a very brief overview of timed automata and their semantics, and of the UPPAAL tool, which is a model checker for timed automata. The reader is referred to [5, 8, 16, 19] for more details.

Timed automata are finite automata that are extended with real valued clock variables [5, 8]. The basic definitions concerning timed automata from [5] are reused here. Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the grammar $\phi := x \sim c \,|\, \phi_1 \wedge \phi_2$, where $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. A clock interpretation ν for a set X is a mapping from X to $\mathbb{R}^+$, where $\mathbb{R}^+$ denotes the set of positive real numbers including zero. A clock interpretation ν for X satisfies a clock constraint ϕ over X, denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν. For $\delta \in \mathbb{R}^+$, $\nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For a set $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$ and agrees with ν over the rest of the clocks. We let $\Gamma(X)$ denote the set of all clock interpretations for X.

A timed automaton then is a tuple $(L, l^0, \Sigma, X, I, E)$, where L is a finite set of locations, $l^0 \in L$ is the initial location, Σ is a finite set of labels, X is a finite set of clocks, I is a mapping that labels each location $l \in L$ with some clock constraint in $\Phi(X)$ (the *location invariant*) and $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges. An edge $(l, a, \phi, \lambda, l')$ represents a transition from location l to location l' on the symbol a. The clock constraint ϕ specifies when the edge is enabled and the set $\lambda \subseteq X$ gives the clocks to be reset with this edge. The semantics of a timed automaton $(L, l^0, \Sigma, X, I, E)$ is defined by associating a transition system with it. A state is a pair (l, ν), where $l \in L$, and $\nu \in \Gamma(X)$ such that $\nu \models I(l)$. The initial state is (l^0, ν^0), where $\nu^0(x) = 0$ for all $x \in X$. There are two types of transitions (let $\delta \in \mathbb{R}^+$ and let $a \in \Sigma$). First, $((l, \nu), (l, \nu + \delta))$ is a *δ-delay transition* iff $\nu + \delta' \models I(l)$ for all $0 \leq \delta' \leq \delta$. Second, $((l, \nu), (l', \nu'))$ is an *a-action transition* iff an edge $(l, a, \phi, \lambda, l')$ exists such that $\nu \models \phi$, $\nu' = \nu[\lambda := 0]$ and $\nu' \models I(l')$. Note that location invariants can be used to specify progress, and that they can cause time deadlocks.

The transition system of a timed automaton is infinite due to the real valued clocks. The region and zone constructions, however, are finite abstractions that preserve Timed Computation Tree Logic (TCTL) formulas and a subset of TCTL formulas (most notably reachability) respectively [6, 44]. This enables the application of finite state model checking techniques as implemented, for instance, by the UPPAAL tool.

The UPPAAL modeling language extends the basic timed automata as defined above with bounded integer variables and binary blocking (CCS style) synchro-

nization. Systems are modeled as a set of communicating timed automata. The UPPAAL tool supports simulation of the model and the verification of reachability and invariant properties. The question whether a state satisfying ϕ is reachable can be formalized as $\mathbf{EF}(\phi)$. The question whether ϕ holds for all reachable states is formalized as $\mathbf{AG}(\phi)$. If a reachability property holds or an invariant property does not hold, then UPPAAL can provide a run that proves this. This run can be replayed in the simulator, which is very useful for debugging purposes.

7.3 Description of the Algorithm

This section presents an informal description of an algorithm that solves the problem of *fault-tolerant distributed agreement* in a partially synchronous setting [10] (see also Chapter 25 of [78]). A system of n processes, denoted by $p_1, ..., p_n$, is considered, where each process is given an input value and at most f processes may fail. Each process that does not fail must eventually (termination) choose a decision value such that no two processes decide differently (agreement), and if any process decides for v, then this has been the input value of some process (validity)[1]. The process's computation steps are atomic and take no time, and two consecutive computation steps of a non-faulty process are separated c_1 to c_2 time units. The processes can communicate by sending messages to each other. The message delay is bounded by d time units, and message delivery is unordered. Furthermore, messages can get neither lost nor duplicated. The constant D is defined as $d + c_2$. As mentioned above, f out of the n processes may fail. A failure may occur at any time, and if a process fails at some point, then an arbitrary subset of the messages that would have been sent in the next computation step, is sent. No further messages are sent by a failed process. It is convenient to regard the algorithm, which is run by every process, as the merge of a *timeout task* and a *main task*, such that a process's computation step consists of a step of the timeout task followed by a step of the main task.

7.3.1 Description of the Timeout Task

The goal of the timeout task is to maintain the running state of all other processes. To this end, every process p_j broadcasts an $(alive, j)$ message in every computation step. If process p_i has run for sufficiently many computation steps without receiving an $(alive, j)$ message, then it assumes that p_j halted either by decision or by failure[2]. Figure 7.1 contains the description of a computation step of the timeout task of process p_i in precondition-effect style.

[1] This is required to avoid trivial solutions in which every process always decides for some predetermined constant value.

[2] The message complexity of this algorithm is quite high. Recently, an alternative with an adjustable "probing load" for each node has been proposed in [20], further analyzed in [72], and improved in [21].

Precondition:
$\neg$ *blocked*
Effect:
broadcast((*alive,i*))
for $j := 1$ **to** n **do**
counter$(j) :=$ *counter*$(j) + 1$
if (*alive,j*) $\in$ *buff* **then**
remove (*alive,j*) from *buff*
counter$(j) := 0$
else if *counter*$(j) \geq \lfloor \frac{D}{c_1} \rfloor + 1$
then
add j to *halted*
od

Figure 7.1: The timeout task for process p_i.

The boolean variable *blocked* is used by the main task to stop the timeout task. Initially, this boolean is *false*. It is set to *true* if the process decides. The other state components are a set *halted* $\subseteq \{1, ..., n\}$, initially $\emptyset$, and for every $j \in \{1, ..., n\}$ a counter *counter*(j), initially set to -1. Additionally, every process has a message buffer *buff* (a set), initially $\emptyset$. Two properties of the timeout task have been proved in [10].

A_1 If any p_i adds j to *halted* at time t, then p_j halts, and every message sent from p_j to p_i is delivered strictly before time t.

A_2 If p_j halts at time t, then every p_i either halts or adds j to *halted* by time $t + T$, where $T = D + c_2 \cdot (\lfloor \frac{D}{c_1} \rfloor + 1)$.

These two properties are used in [10] for the correctness proof of the complete algorithm. In this paper, these two properties are first mechanically verified for a number of instances of the algorithm. Consequently, they are used to make an abstract model of the complete algorithm in Section 7.5.

7.3.2 Description of the Main Task

Figure 7.2 contains the description of a computation step of the main task of process p_i in precondition-effect style. Apart from the input value v_i and the state components used by the timeout task, there is one additional state component, namely the round counter r, initially zero. The input values are assumed to be either zero or one for simplicity[3].

Each process tries to decide in each round. Note that a process may decide for 0 only in even rounds, and for 1 only in odd rounds. Furthermore, if a process fails

[3] An extension to an arbitrary input domain is discussed in [10].

Precondition:
$r = 0 \wedge v_i = 1$
Effect:
broadcast$((0,i))$
$r := 1$

Precondition:
$r \geq 1 \wedge \exists_j\ (r,j) \in$ *buff*
Effect:
broadcast$((r,i))$
$r := r + 1$

Precondition:
$r = 0 \wedge v_i = 0$
Effect:
broadcast$((1,i))$
decide(0)

Precondition:
$r \geq 1 \wedge \forall_{j \notin \textit{halted}}\ (r-1,j) \in$ *buff* $\wedge$
$\neg\exists_j\ (r,j) \in$ *buff*
Effect:
broadcast$((r+1,i))$
decide$(r \bmod 2)$

Figure 7.2: The main task for process p_i.

to decide in round r, then it broadcasts r before going to round $r+1$. On the other hand, if a process decides in round r, it broadcasts $r+1$ before halting. In order for a process to decide in a round $r \geq 1$, it ensures that it has received the message $r-1$ from all non-halted processes, and no message r from any process. Three main results that are obtained in [10] are the following.

M_1 (Agreement, Lemma 5.9 of [10]). No two processes decide on different values.

M_2 (Validity, Lemma 5.10 of [10]). If process p_i decides on n, then $n = v_j$ for some process j.

M_3 (Termination, Theorem 5.1 of [10]). The upper bound on the time to reach agreement equals $(2f-1)D + max\ \{T, 3D\}$.

These results are mechanically verified in Section 7.5 for a number of non-trivial instances of the algorithm.

7.4 Verification of the Timeout Task

7.4.1 Modeling the Timeout Task

Note that every process runs the same algorithm, and that the timeout parts of different processes do not interfere with each other. Therefore, only two processes are considered, say p_i and p_j. By the same argument, only one direction of the timeout task is considered: p_i (*Observer*) keeps track of the running state of p_j (*Process*).

Figure 7.3 shows the UPPAAL automaton of the merge of the timeout task and abstract main task of *Process* (the only functionality of the main task is to halt). It has one local clock x to keep track of the time between two consecutive computation steps. The *Process* automaton must spend exactly c_2 time units in the

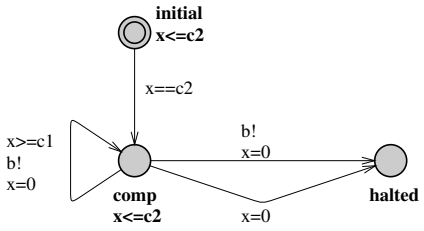

Figure 7.3: The *Process* automaton.

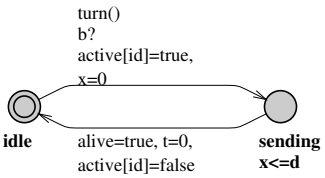

Figure 7.4: The broadcast template.

initial location *init* before it takes the transition to location *comp* (the reason for this is explained below). It then immediately either fails or does a computation step. Failure of *Process* is modeled by the pair of edges to *halted*, which models the non-deterministic choice of the subset of messages to send. The computation step is modeled by the self-loop and by the upper transition to *halted* (a decision transition that blocks the timeout task)[4]. Note that x is reset on every edge to *halted* for verification purposes.

As required by the algorithm, *Process* broadcasts an *alive* message at each computation step. This action is modeled by a b-synchronization, which activates an instance of the broadcast template, shown in Figure 7.4. This template is parameterized with a constant *id* in order to give each instance a unique identifier. Clearly, the UPPAAL model must ensure *output enabledness* of *Process*: it must be able to broadcast the alive message when it wants to. Since the maximal number of simultaneous broadcasts equals $\lfloor \frac{d}{c_1} \rfloor + 2$, this many instances of the broadcast template must be present in the model. The guard *turn()* and the assignments to *active[id]* implement a trick to reduce the reachable state space by partially exploiting the symmetry among the broadcast instances[5]. After a b-synchronization, a broadcast automaton may spend at most d time units in location *sending*, which is modeled using the local clock x. The actual message delivery is modeled by the assignment *alive=true* on the transition back to *idle*. The reset of the global clock t is used for the verification of property A_1.

Figure 7.5 shows the automaton for the *Observer*, which is the composition of an abstract main task (whose only purpose again is to halt) and the "receiving part" of the timeout task. It has a local integer variable *cnt*, initialized to -1, and a local clock x. Furthermore, the boolean *has_halted* models whether $Process \in halted_{Observer}$. The *Observer* automaton must first spend c_2 time units in the initial location before taking the edge to location *comp*. Then, it must immediately either do a computation step or fail. The computation step is modeled by the self-loop and by the upper transition to *halted*. The procedure *update()* updates the variables

[4] A straightforward model contains a third edge to *halted* with the guard $x \geq c_1$, the synchronization *b!*, and the reset $x = 0$. Such an edge is, however, "covered" by the present upper edge to *halted* and can therefore be left out

[5] A next release of UPPAAL will hopefully support symmetry reduction, which can automatically exploit the symmetry among broadcast automata [57].

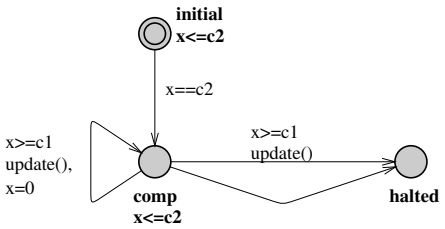

Figure 7.5: The *Observer* automaton.

```
void update ()
{
  if (!has_halted)
    cnt++;
  if (alive)
  {
    alive = false;
    cnt = 0;
  }
  has_halted = cnt>=(D/c1)+1;
}
```

Figure 7.6: The *update()* function.

cnt, *has_halted* and *alive* as specified in Figure 7.6. Failure is modeled by the lower edge to *halted*.

Both the *Observer* automaton and the *Process* automaton must first spend c_2 time units in their initial location. This is a modeling trick to fulfill the requirement from [10] that "every process has a computation or failure event at time 0". I.e., our model starts at time $-c_2$. (If UPPAAL would allow the initialization of a clock to any natural number, then both initial locations can be removed.)

7.4.2 Verifying the Timeout Task

Property A_1 is translated to the following invariant property of the UPPAAL model (a broadcast automaton with identifier i is denoted by b_i):

$$\mathbf{AG}\left(\begin{array}{c} \mathit{has\_halted} \longrightarrow \\ (\mathit{Process.halted} \wedge \forall_i\, b_i.\mathit{idle} \wedge t > 0) \end{array}\right) \tag{7.1}$$

The state property $\forall_i\, b_i.\mathit{idle} \wedge t > 0$ ensures that all messages from *Process* to *Observer* are delivered strictly before the conclusion of *Observer* that *Process* halted. Property A_2 is translated as follows:

$$\mathbf{AG}\left(\begin{array}{c} (\mathit{Process.halted} \wedge \mathit{Process.x} > T) \\ \longrightarrow \\ (\mathit{Observer.halted} \vee \mathit{has\_halted}) \end{array}\right) \tag{7.2}$$

The branching time nature of A_2 is specified by this invariance property due to the structure of our model: $\mathit{Process.x}$ measures the time that has been elapsed since *Process* arrived in the location *halted*.

Properties (7.1) and (7.2) have been verified for the following parameter values[6]:

- $c_1 = 1, c_2 = 1$ and $d \in \{0 - 5\}$,

[6] A 3.4 GHz Pentium 4 machine with 2 GB of main memory running Fedora Core 4 has been used for all measurements. The tool *memtime* (available via the UPPAAL website `http://www.uppaal.com/`) has been used to measure the time and memory consumption.

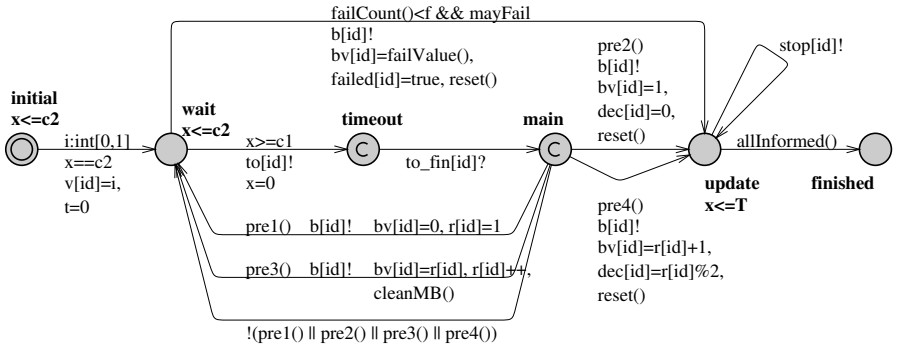

Figure 7.7: The process template.

- $c_1 = 1$, $c_2 = 2$ and $d \in \{0 - 5\}$, and
- $c_1 = 9$, $c_2 = 10$ and $d \in \{5, 9 - 11, 15, 20, 50\}$.

Each of the above instances could be verified within 5 minutes using at most 25 MB of memory.

7.5 Verification of the Algorithm

The UPPAAL model of the parallel composition of the main task and the timeout task, which is used to verify properties M_1–M_3, is presented in this section. It is assumed that every process receives an input by time zero (synchronous start), since otherwise the state space becomes too large to handle interesting instances. If the timeout task is modeled explicitly, then many *alive* messages must be sent every computation step, which results in an overly complex model. Using properties A_1 and A_2, however, the explicit sending of *alive* messages can be abstracted away.

7.5.1 Modeling the Algorithm

Figure 7.7 shows the UPPAAL template of the behavior of the algorithm. This template is parameterized with two constants, namely its unique identifier *id*, and a boolean *mayFail* which indicates whether this process may fail[7].

Similar to the model of the timeout task, a process first waits c_2 time units in its initial location. Then, it non-deterministically chooses an input value in $\{0, 1\}$ on the edge to *wait*. The global clock t is used to measure the running time of the algorithm, and is only reset on this edge. Then it either starts a computation step or fails. A computation step first activates the timeout automaton of the process, which is described below, on the edge to *timeout*. When the timeout automaton finishes (it may have updated the *halted* set), the edge to *main* is taken. Then there

[7] Again, this is a trick that exploits the symmetry of processes to reduce the reachable state space.

```
bool pre1 ()
{
  return r[id]==0 && v[id]==1;
}

bool pre3 ()
{
  if (r[id]<=0)
    return false;
  for (j:pid_t)
    if (buff[id][r[id]][j])
      return true;
  return false;
}
```

```
bool pre2 ()
{
  return r[id]==0 && v[id]==0;
}

bool pre4 ()
{
  if (r[id]<=0 || pre3())
    return false;
 for (j:pid_t)
    if (!halted[id][j] &&
        !buff[id][r[id]-1][j])
      return false;
  return true;
}
```

Figure 7.8: The preconditions for the four transitions of the main task.

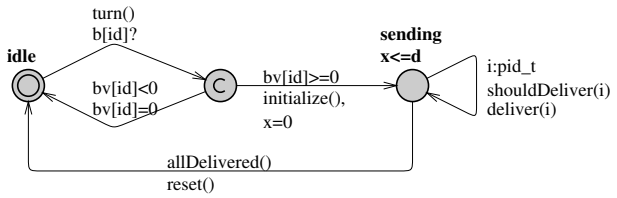

Figure 7.9: The broadcast template.

are five possibilities: one of the four preconditions of the main task transitions is satisfied (note that they are all mutually exclusive), or none of them is satisfied. In the first case, the specified actions are taken, and in the second case nothing is done. The committed locations (those with a "C" inside) specify that a computation step is atomic and that it takes no time (if a committed location is active, then no delay is allowed and the next action transition must involve a committed component). Note that broadcasting the message (m, i) is achieved by assigning m to *bv[id]* on an edge with a *b[id]*-synchronization. Figure 7.8 shows the functions that implement the preconditions of the four transitions of the main task (see also Figure 7.2).

A failure is modeled by the edge from *wait* to *update*. This edge is only enabled if fewer than f failures already have occurred. The *failValue()* function computes the value that would have been broadcast during the next computation step.

In location *update* the process has halted either by decision or by failure. It can stay there for a maximum of T time units and during that time it provides a *stop[id]*-synchronization. This is used for the abstraction of the timeout task, which is explained below. When all other processes have been informed that this process has halted (*allInformed()* returns *true*), then the transition to location *finished* is enabled.

Similar to the model of the timeout task, the broadcasts are modeled by instances of the broadcast template which is shown in Figure 7.9.

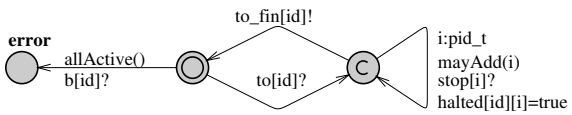

Figure 7.10: The timeout template.

The template is parameterized with two constants, namely *id*, the identifier of the process automaton this broadcast automaton belongs to, and *bid*, an identifier that is unique among the other broadcast automata of process automaton *id*. The broadcast automaton is started – if it is its turn[8] – with a *b[id]*-synchronization. If the value of *bv[id]* is smaller than zero, then nothing is done (this is convenient for modeling in the process template). In location *sending* it starts delivering the message that has been passed to it in *bv[id]*. The *shouldDeliver()* and *allDelivered()* functions ensure that it delivers all messages on time, but only if necessary. I.e., it is not useful to deliver a message to a process that already has halted, since that message is never used; it only increases the reachable state space.

Each process automaton has a separate timeout automaton that has two functions. First, it is activated at the beginning of each computation step of the process it belongs to in order to update the *halted* set of the process. Second, it serves as a test automaton to ensure that the process it belongs to is output enabled[9]. The timeout template is shown in Figure 7.10. It has one parameter, namely the constant *id*, which refers to the process it belongs to.

When a timeout process is activated, it non-deterministically picks a subset of processes that have halted and adds them to the *halted* set. Here properties A_1 and A_2 of the timeout task come in. The function *mayAdd()* checks for a given process j whether all messages from j to this process have been delivered. If not, then it may not add j to *halted* (property A_1). Furthermore, the synchronization over the channel *stop[j]* must be enabled. In Figure 7.7 can be seen that this is only the case for the T time units after j has halted (property A_2). But if this process has not added j to halted by that time, then j cannot proceed to location *finished* (in that case *allInformed()* returns *false*), with a time deadlock as result. This is exactly the case when $T - p_i.x < c_1 - p_j.x$ for processes i and j. We believe that this abstraction of the timeout task is safe, i.e., every admissible computation path in the original model of [10] can be mapped to an equivalent path in the UPPAAL model.

The second function of the timeout template is implemented by the edge to the *error* location. This location is reachable if the process wants to broadcast and all its broadcast automata are active already. In a correct model, the *error* location

[8] Similarly as in the model of the timeout task in the previous section, the guard *turn()* partially exploits the symmetry between the broadcast automata of a single process to reduce the reachable state space.

[9] In this model, the number of necessary broadcast automata is no longer easily to determine. Therefore, an explicit check is useful.

therefore is not reachable.

7.5.2 Verifying the Algorithm

Properties M_1–M_3 are translated as follows (where U is the upper bound on the running time of the protocol as specified before).

$$\textit{Agreement:} \quad \mathbf{AG}\Big(\forall_{i,j}\ \textit{dec}_i \geq 0 \wedge \textit{dec}_j \geq 0 \longrightarrow \textit{dec}_i = \textit{dec}_j\Big) \tag{7.3}$$

$$\textit{Validity:} \quad \mathbf{AG}\Big(\forall_i\ \textit{dec}_i \geq 0 \longrightarrow \exists_j\ \textit{dec}_i = v_j\Big) \tag{7.4}$$

$$\textit{Termination:} \quad \mathbf{AG}\Big((\exists_i\ p_i.\textit{wait}) \longrightarrow t \leq U\Big) \tag{7.5}$$

The following properties are health checks to ensure that (i) the processes are output enabled, and (ii) the only deadlocks in the model are those that are expected.

$$\mathbf{AG}\Big(\neg\exists_i\ T_i.\textit{error}\Big) \tag{7.6}$$

$$\mathbf{AG}\ \big(\ \textit{deadlock} \longrightarrow (\forall_i\ p_i.\textit{finished}\ \vee \exists_{i,j}\ p_j.x - p_i.x > T - c_1)\ \big) \tag{7.7}$$

The properties (7.3)–(7.6) have been verified (using the convex-hull approximation of UPPAAL with a breadth-first search order) for the following parameter values[6]:

- $n = 3$, $f \in \{0, 1\}$, $c_1 = 1$, $c_2 = 1$, and $d \in \{0, 1, 2, 3, 5, 10\}$,
- $n = 3$, $f \in \{0, 1\}$, $c_1 = 1$, $c_2 = 2$, and $d \in \{0, 1, 2, 3, 5, 10\}$, and
- $n = 3$, $f \in \{0, 1\}$, $c_1 = 9$, $c_2 = 10$, and $d \in \{5, 9 - 11, 15, 20, 50, 100\}$.

Each of the above instances could be verified within 11 minutes using at most 1014 MB of memory. Property (7.7) has been verified for a subset of the above parameter values, namely for the models with the three smallest values for d in each item. This property is more difficult to model check since the convex-hull approximation is not useful and it involves the *deadlock* state property, which disables UPPAAL's LU-abstraction algorithm [14] (a less efficient one is used instead), and which is computationally quite complex due to the symbolic representation of states.

7.6 Conclusions

Despite the fact that model checkers are in general quite easy to use (in the sense that their learning curve is not so steep as for instance the one of theorem provers), making a good model still is difficult. The algorithm that has been analyzed in this paper can easily be modeled "literally". The message complexity then, however, is huge due to the many broadcasts of *alive* messages, with the result that

model checking interesting instances becomes impossible. This has been solved by a non-trivial abstraction of the timeout task. Ideally of course, model checkers can even handle such "naive" models. Fortunately, much research still is aimed at improving these tools. For instance, the UPPAAL model checker is getting more and more mature, both w.r.t. usability as efficiency. An example of the former is the recent addition of a C-like language. This makes the modeling of the agreement protocol much easier, and makes the model more efficient. A loop over an array, as for instance used in the *pre3()* and *pre4()* functions shown in Figure 7.8, can now be encoded with a C-like function instead of using a cycle of committed locations and/or an auxiliary variable. This saves the allocation and deallocation of intermediate states and possibly a state variable. Other examples of efficiency improvements of UPPAAL are enhancements like symmetry reduction [57] and the sweep line method [36], which are planned to be added to UPPAAL soon. Especially symmetry reduction would greatly benefit distributed systems, which often exhibit full symmetry. Furthermore, recent research also focuses on distributing UPPAAL, which may even give a super-linear speed-up [17, 12].

It seems that the class of partially synchronous systems, which is notoriously difficult from the perspective of model checking, now slowly comes within reach of present model checking tools. Therefore, these tools have the potential to play a valuable role in the design of these systems. They may provide valuable early feedback on subtle design errors and hint at system invariants that can subsequently be used in the general correctness proof.

Acknowledgements. The author thanks Frits Vaandrager and Jozef Hooman for valuable discussions and comments on earlier versions of the present paper.

Bibliography

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.

[2] Y. Abdeddaïm, A. Kerba, and O. Maler. Task graph scheduling using timed automata. In *8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)*, 2003.

[3] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, number 2102 in LNCS, pages 478–492. Springer–Verlag, 2001.

[4] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parameterized system verification. In *11th International Conference on Computer Aided Verification*, number 1633 in LNCS, pages 134–145. Springer–Verlag, 1999.

[5] R. Alur. Timed automata. In *11th International Conference on Computer Aided Verification*, number 1633 in LNCS, pages 8–22. Springer–Verlag, 1999.

[6] R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[7] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages and Programming*, pages 322–335, 1990.

[8] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[9] European Community Project IST-2001-35304 AMETIST (Advanced Methods for Timed Systems). `http://ametist.cs.utwente.nl/`.

[10] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, 1994.

[11] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, number 2057 in LNCS, pages 103–122. Springer, 2001.

[12] G. Behrmann. Distributed reachability analysis in timed automata. *Software Tools for Technology Transfer*, November 2003. Issue: Online First.

[13] G. Behrmann, P. Bouyer, E. Fleury, and K. G. Larsen. Static guard analysis in timed automata verification. In H. Garavel and J. Hatcliff, editors, *TACAS 2003*, number 2619 in LNCS, pages 254–270. Springer–Verlag, 2003.

[14] G. Behrmann, P. Bouyer, K. G. Larsen, and R. Pelánek. Lower and upper bounds in zone based abstractions of timed automata. In K. Jensen and A. Podelski, editors, *TACAS'04*, number 2988 in LNCS, pages 312–326. Springer–Verlag, 2004.

[15] G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader. Scheduling lacquer production by reachability analysis – a case study. In *Workshop on Parallel and Distributed Real-Time Systems 2005*. IEEE Computer Society Press, 2005. Full version, to appear.

[16] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *SFM'04*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004. *http://www.uppaal.com/*.

[17] G. Behrmann, T. Hune, and F. W. Vaandrager. Distributed timed model checking – how the search order matters. In E.A. Emerson and A.P. Sistla, editors, *CAV'2000*, number 1855 in LNCS, pages 216–231. Springer–Verlag, 2000.

[18] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[19] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, number 3098 in LNCS, pages 87–124. Springer–Verlag, 2004.

[20] M. Bodlaender, J. Guidi, and L. Heerink. Enhancing discovery with liveness. In *CCNC'04*. IEEE Computer Society Press, January 2004.

[21] H. Bohnenkamp, J. Gorter, J. Guidi, and J.-P. Katoen. Are you still there? A lightweigth algorithm to monitor node absence in self-configuring networks. In *Proceedings of DSN 2005*, 2005.

[22] H. C. Bohnenkamp, H. Hermanns, R. Klaren, A. Mader, and Y. S. Usenko. Synthesis and stochastic assessment of schedules for lacquer production. In *Proceedings 1st International Conference on Quantitative Evaluation of Systems (QEST 2004),* 27-30 September 2004, Enschede, The Netherlands, pages 28–37. IEEE Computer Society, 2004.

[23] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *8th International Conference on Computer Aided Verification*, number 1102 in LNCS, pages 1–12. Springer–Verlag, 1996.

[24] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *4th International Static Analysis Symposium*, LNCS. Springer–Verlag, 1997.

[25] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *6th International Conference on Computer Aided Verification*, number 808 in LNCS, pages 55–67. Springer–Verlag, 1994.

[26] D. Bosnacki, D. Dams, and L. Holenderski. A heuristic for symmetry reductions with scalarsets. In J.N. Oliveira and P. Zave, editors, *FME 2001*, number 2021 in LNCS, pages 518–533. Springer–Verlag, 2001.

[27] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with non-regular sets of configurations. In *24th International Colloquium on Automata, Languages, and Programming*, number 1256 in LNCS. Springer–Verlag, 1997.

[28] M. Bozga and O. Maler. Timed automata approach for the AXXOM case study, 2003. Available through the URL `http://www-verimag.imag.fr/~maler/AMETIST/axxom-report.pdf`.

[29] V. Braberman, D. Garbervetsky, and A. Olivero. Obsslice: A timed automata slicer based on observers. In *16th Conference on Computer-Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 470–474. Springer, 2004.

[30] N. C. W. M. Braspenning. Scheduling and behavior verification of machines based on task-resource models. Master's thesis, Department of Mechanical Engineering, Eindhoven University of Technology, The Netherlands, October 2003. Confidential.

[31] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1,2):115–131, 1988.

[32] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems: Advanced Lectures*. Number 3472 in LNCS. Springer, 2005.

[33] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, August 1986.

[34] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses, 2004. *http://www.ietf.org/internet-drafts/draft-ietf-zeroconf-ipv4-linklocal-17.txt*.

[35] D. Chkliaev, J. Hooman, and E. de Vink. Verification and improvement of the sliding window protocol. In *TACAS'03*, number 2619 in LNCS, pages 113–127. Springer–Verlag, 2003.

[36] A. Christensen, L. M. Kristensen, and T. Mailund. A sweep-line method for state space exploration. In *TACAS'01*, number 2031 in LNCS, pages 450–464. Springer–Verlag, April 2001.

[37] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[38] E. M. Clarke, S. Jha, R. Enders, and T. Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods in System Design*, 9(1/2):77–104, 1996.

[39] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from JAVA source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000. Available through URL `http://www.cis.ksu.edu/santos/bandera/`.

[40] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *17th IEEE Real Time Systems Symposium (RTSS'96)*. IEEE CS Press, 1996.

[41] H. Dierks. *Specification and Verification of Polling Real-Time Systems*. PhD thesis, Carl von Ossietzky Universität Oldenburg, July 1999.

[42] E. W. Dijkstra. Cooperating sequential processes. Technical report, Eindhoven University of Technology, The Netherlands, 1965.

[43] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *International Workshop on Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 197–212. Springer–Verlag, 1989.

[44] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, number 407 in LNCS, pages 197–212. Springer–Verlag, 1989.

[45] D. L. Dill, A. J. Drexler, A. J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525. IEEE Computer Society, 1992.

[46] M. B. Dwyer, J. Hatcliff, M. Hoosier, and Robby. Building your own software model checker using the bogor extensible model checking framework. In *17th Conference on Computer-Aided Verification (CAV'05)*, volume 3576 of *LNCS*, pages 148–152. Springer, 2005.

[47] L. Elgaard. *The Symmetry Method for Coloured Petri Nets - Theory, Tools, and Practical Use*. PhD thesis, Department of Computing Science, University of Aarhus, Denmark, July 2002.

[48] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *CAV'93*, number 697 in LNCS. Springer–Verlag, 1993.

[49] A. Fehnker. Scheduling a steel plant with timed automata. In *Proceedings of the sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*. IEEE Computer Society Press, 1999.

[50] B. Gebremichael and F. W. Vaandrager. Control synthesis for a smart card personalization system using symbolic model checking. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, number 2791 in LNCS, pages 189–203. Springer–Verlag, 2004.

[51] B. Gebremichael, M. Zhang, and F. W. Vaandrager. Analysis of a protocol for dynamic configuration of IPv4 link local addresses using Uppaal. Technical Report ICIS-R06XXX, ICIS, Radboud University Nijmegen, 2006. To appear.

[52] V. Hartonas-Garmhausen, E. M. Clarke, and S. Campos. Deadlock prevention in flexible manufacturing systems using symbolic model checking. In *IEEE Conference on Robotics and Automation*, volume 1, pages 527–532, 1996.

[53] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *18th IEEE Real-Time Systems Symposium*, pages 2–13, 1997.

[54] M. Hendriks. Development of reactive programs using UPPAAL. Master's thesis, University of Nijmegen, The Netherlands, February 2002. Available through URL *http://www.cs.kun.nl/ita/publications/mastertheses/200202_martijn_hendriks/*.

[55] M. Hendriks. Enhancing UPPAAL by exploiting symmetry. Technical Report NIII-R0208, NIII, University of Nijmegen, October 2002.

[56] M. Hendriks. Model checking the time to reach agreement. In P. Pettersson and W. Yi, editors, *3rd International Conference on the Formal Modeling and Analysis of Timed Systems (FORMATS'05)*, number 3829 in LNCS, pages 98–111. Springer–Verlag, 2005.

[57] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding symmetry reduction to Uppaal. In K. G. Larsen and P. Niebert, editors, *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, number 2791 in LNCS, pages 46–59. Springer–Verlag, 2004.

[58] M. Hendriks, G. Behrmann, K.G. Larsen, P. Niebert, and F.W. Vaandrager. Adding symmetry reduction to UPPAAL. Technical Report NIII-R0407, NIII, University of Nijmegen, February 2004.

[59] M. Hendriks and K. G. Larsen. Exact acceleration of real-time model checking. In E. Asarin, O. Maler, and S. Yovine, editors, *Workshop on Theory and Practice of Timed Systems (TPTS'02)*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2002.

[60] M. Hendriks, N. J. M. van den Nieuwelaar, and F. W. Vaandrager. Model checker aided design of a controller for a wafer scanner. 2006. To appear in a special issue of Software Tools for Technology Transfer.

[61] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. 2006. Accepted for the 14th International Workshop on Parallel and Distributed Real-Time Systems (WPDRTS'06).

[62] T. A. Henzinger, P. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.

[63] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *10th SPIN Workshop on Model Checking Software (SPIN'03)*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[64] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[65] G. J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[66] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Reachability trees for high-level Petri nets. *Theoretical Computer Science*, 45(3):261–292, 1986.

[67] T. S. Hune. Modeling a language for embedded systems in timed automata. Technical Report RS-00-17, BRICS, Basic Research in computer Science, August 2000. 26 pp. Earlier version entitled *Modelling a Real-Time Language* appeared in FMICS99, pages 259–282.

[68] C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Computer Hardware Description Languages and their Applications*, pages 87–100, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, The Netherlands. Journal version appeared in *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[69] C. N. Ip and D. L. Dill. Efficient verification of symmetric concurrent systems. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 230–234, Cambridge, MA, 1993.

[70] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, M. Laursen, R. G. Madsen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-checking real-time control programs — verifying LEGO mindstorms systems using UPPAAL. In *IEEE Euromicro Conference on Real-Time Systems*, pages 147–155, 2000.

[71] K. Jensen. Condensed state spaces for symmetrical Coloured Petri Nets. *Formal Methods in System Design*, 9(1/2):7–40, 1996.

[72] J.-P. Katoen, H. Bohnenkamp, R. Klaren, and H. Hermanns. Embedded software analysis with MOTOR. In *SFM'04*, number 3185 in LNCS, pages 268–293. Springer–Verlag, 2004.

[73] D. K. Kaynar, N. A. Lynch, R. Segala, and F. W. Vaandrager. A framework for modelling timed systems with restricted hybrid automata. In *RTSS'03*, pages 166–177. IEEE Computer Society Press, 2003. A full version is available as MIT Technical Report MIT/LCS/TR-917.

[74] L. Lamport. Real-time model checking is really simple. In D. Borrione and W. Paul, editors, *CHARME 2005*, LNCS. Springer, 2005. To appear.

[75] K. G. Larsen, G. Behrmann, E. Brinksma, A. Fehnker, T. Hune, P. Pettersson, and J. Romijn. As cheap as possible: Efficient cost-optimal reachability for priced timed automata. In G. Berry, H. Comon, and A. Finkel, editors, *CAV 2001*, number 2102 in LNCS, pages 493–505. Springer–Verlag, 2001.

[76] M. Lawley and S. A. Reveliotis. Deadlock avoidance for sequential resource allocation systems: Hard and easy cases. *International Journal of Flexible Manufacturing Systems*, 13(4):385–404, 2001.

[77] M. Lawley, S. A. Reveliotis, and P. Ferreira. Design guidelines for deadlock handling strategies in flexible manufacturing systems. *International Journal of Flexible Manufacturing Systems*, 9(1):5–30, January 1997.

[78] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

[79] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC'87*, pages 137–151, 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.

[80] N. A. Lynch and M. R. Tuttle. An introduction to Input/Output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[81] A. Mader. Deriving schedules for the CYBERNETIX case study, 2003. Available via the URL `http://ametist.cs.utwente.nl/RESEARCH/TWENTE/ANGELIKA/cybernetix_angelika.ps`.

[82] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proceedings of STACS'95*, volume 900 of *LNCS*. Springer-Verlag, 1995.

[83] K. L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, Pittsburgh, May 1992.

[84] M. Menasche and B. Berthomieu. Time petri nets for analyzing and verifying time dependent communication protocols. In H. Rudin and C. H. West, editors, *Protocol Specification, Testing and Verification III*, 1983.

[85] M. O. Möller. Parking can get you there faster. In *Proceedings of the Workshop on Theory and Practice of Timed Systems*, volume 65 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, April 2002.

[86] T. Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, 1989.

[87] P. Niebert and S. Yovine. Computing optimal operation schemes for chemical plants in multi-batch mode. In *HSCC*, volume 1790 of *LNCS*, pages 338–351. Springer, 2000.

[88] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[89] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer–Verlag, 1992.

[90] J. Park and S. A. Reveliotis. Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings. *IEEE Transactions on Automatic Control*, 46(10):1572–1583, 2001.

[91] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 2002.

[92] M. Pinedo and X. Chao. *Operations Scheduling with Applications in Manufacturing Systems*. McGraw-Hill, 1999.

[93] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *12th International Conference on Computer Aided Verification*, number 1855 in LNCS, pages 328–343. Springer–Verlag, 2000.

[94] S. A. Reveliotis, M. Lawley, and P. Ferreira. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Transactions on Automatic Control*, 42(10):1344–1357, 1997.

[95] B. Schtz, Pretschner A., F. Huber, and J. Philipps. Model-based development of embedded systems. In J.-M. Bruel and Z. Bellahsène, editors, *Advances in Object-Oriented Information Systems*, volume 2426 of *LNCS*, pages 298–311. Springer, 2002.

[96] J. Sifakis, S. Tripakis, and S. Yovine. Building models of real-time systems from application software. *Proceedins of the IEEE*, 91(1):100–111, 2003.

[97] W. Stallings. *Operating Systems – Internals and Design Principles*. Prentice–Hall, 1998.

[98] P. H. Starke. Reachability analysis of Petri nets using symmetries. *Syst. Anal. Model. Simul./5*, 8(4):293–303, 1991.

[99] A. S. Tanenbaum. *Computer Networks*. Prentice–Hall, 1996.

[100] S. Tripakis. *The analysis of timed systems in practice*. PhD thesis, Joseph Fourier University, Grenoble, France, December 1998.

[101] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10:203–232, 2003.

[102] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In S. Graf and M. Schwartzbach, editors, *TACAS'00*, number 1785 in LNCS, pages 157–171. Springer–Verlag, 2000.

[103] F. Wang. Efficient verification of timed automata with BDD-like data structures. *Software Tools for Technology Transfer*, 6(1):77–97, 2004.

[104] F. Wang and K. Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In D.A. Peled and M.Y. Vardi, editors, *FORTE'02*, number 2529 in LNCS, pages 50–64. Springer–Verlag, 2002.

[105] Y. Wang and Z. Wu. Deadlock avoidance control synthesis in manufacturing systems using model checking. In *IEEE American Control Conference*, volume 2, pages 1702–1704, 2003.

[106] K. Yorav. *Exploiting Syntactic Structure for Automatic Verification*. PhD thesis, The Technion, Israel Institute of Technology, Israel, June 2000.

[107] S. Yovine. KRONOS: a verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1/2):123–133, 1997.

Samenvatting (Dutch summary)

Model Checking & Geklokte Automaten

Dit proefschrift gaat over het *model checken* van *geklokte automaten*[10]. Model checken is een techniek om systemen die gemodelleerd zijn in een wiskundige taal automatisch correct te bewijzen. Omdat een formele wiskundige taal wordt gebruikt om systemen te modelleren heeft het model een ondubbelzinnige semantiek (betekenis). Een model checker is een programma dat de toestandsruimte van een door de gebruiker gegeven model automatisch doorzoekt, en controleert of een door de gebruiker gegeven specificatie klopt.

Het geklokte automaten formalisme is uitermate geschikt voor het modelleren van allerlei realistische problemen die met tijd te maken hebben. Typische voorbeelden hiervan zijn gedistribueerde algorithmen, protocollen, embedded software en schedulingproblemen zoals de productiescheduling in fabrieken. Ondanks het feit dat het transitiesysteem van een geklokte automaten model oneindig is (omdat de tijd door de reële getallen wordt gerepresenteerd), bestaan er efficiënte model checkers voor deze modellen zoals KRONOS en UPPAAL.

Model checking is een rigoreuze wiskundige methode die relatief makkelijk te leren en te gebruiken is. Deze techniek heeft echter ook duidelijke nadelen[11]:

1. *Schaalbaarheid.* Model checkers hebben grote last van de zogenaamde "toestandsruimte explosie". Het aantal toestanden van een model groeit exponentieel met het aantal componenten waaruit het is opgebouwd. Dit geeft praktische problemen omdat model checkers in het algemeen de hele toestandsruimte (of een significant deel) in het computergeheugen op moeten slaan. Daarom is het vaak niet mogelijk om realistische problemen op een rechttoe-rechtaan-manier te modelleren en te analyseren.

2. *Toegankelijkheid.* Het bouwen van een goed model is moeilijk. Model checkers zijn vaak academische tools zonder uitgebreide handleiding. Bovendien is een goede kennis van het onderliggende formalisme vereist om modellen te maken die geschikt zijn voor analyse.

3. *Gebruiksvriendelijkheid.* Model checkers zijn zelden onderdeel van de ontwikkelingsomgeving en daarom is er weinig tot geen automatische koppeling met deze tools. Bovendien zijn de algemene invoertalen van model checkers vaak niet uitgebreid genoeg voor een industriële omgeving. Deze twee factoren zorgen ervoor dat het modelleren en analyseren zeer veel tijd vergen.

[10] De Engelse term voor dit formalisme is *timed automata*.

[11] De laatste drie nadelen slaan op algemene tools als KRONOS en UPPAAL en niet op sommige specialistische "in house" industriële tools die van model checking technieken gebruik maken.

4. *Realiseerbaarheid.* Het is zeer vaak onduidelijk wat de relatie tussen een model en de werkelijkheid is. Men kan een zeer abstract ontwerp verifiëren met model checking technieken, maar wat zegt dat over de realisatie van dat ontwerp?

Ondanks deze nadelen zijn model checkers toch zeer nuttig in de praktijk. Vele protocollen zijn reeds geanalyseerd en niet zelden werden fouten opgespoord die niet met conventionele technieken zijn gevonden.

Dit proefschrift

Het doel van dit proefschrift is tweeledig. Ten eerste wordt getracht om nieuwe technieken te ontwikkelen en te implementeren die de toestandsruimteëxplosie van geklokte automaten modellen aanpakken. Dit draagt bij aan het onderzoek naar het *schaalbaarheidsprobleem*. Ten tweede wordt getracht de praktische toepasbaarheid van model chekers voor geklokte automaten te demonstreren en te evalueren. Dit draagt bij aan het onderzoek naar de *toegankelijkheids-* en *gebruiksvriendelijkheidsproblemen*.

Dit proefschrift is geheel tot stand gekomen binnen het kader van het EU project IST-2001-35304 AMETIST (`http://ametist.cs.utwente.nl/`). Het doel van dit project was om de drie eerstgenoemde problemen met betrekking tot model checking op te lossen. Het bovengenoemde doel van dit proefschrift sluit hier dan ook goed bij aan.

Dit proefschrift bestaat uit zes op zichzelf staande artikelen die hieronder summier zijn samengevat:

- Hoofdstuk 2: *Exact Acceleration of Real-Time Model Checking*. Dit hoofdstuk gaat over de toestandsruimteëxplosie ten gevolge van de verschillende "tijdsschalen" in een model. De bijdrage bestaat uit een stelling die dit probleem verkleint voor een subklasse van geklokte automaten.

- Hoofdstuk 3: *Enhancing Uppaal by Exploiting Symmetery*. Symmetriereductie is een bekende techniek om de toestandsruimte van een model te verkleinen wanneer er identieke componenten zijn. De bijdrage van dit werk is een bewijs dat symmetriereductie ook voor geklokte automaten kan worden gebruikt.

- Hoofdstuk 4: *Adding Symmetry Reduction to Uppaal*. Dit hoofstuk laat zien dat de symbolische representatie van tijd de symmetriereductietechniek uit hoofdstuk 3 niet gecompliceerder maakt dan in een situatie zonder tijd. De techniek is in een prototype van het UPPAAL tool verwerkt en laat een exponentiële verbetering zien voor sommige modellen.

- Hoofdstuk 5: *Model Checker Aided Design of a Controller for a Wafer Scanner*. Deze case study laat zien dat een verificatie- en een optimalisatieprobleem op verschillende abstractieniveaus kunnen worden opgelost binnen

één raamwerk met model checking technieken. Dit werk is onderdeel van patentaanvraag ASML ref. P-1784.010 en laat daarmee zijn relevantie voor de industrie zien.

- Hoofdstuk 6: *Production Scheduling by Reachability Analysis*. In deze case study worden geklokte automaten die zijn uitgebreid met kosten functies gebruikt om schedules te vinden voor de productie van lakken. Er wordt aangetoond dat de aanpak gebaseerd op model checking in deze case study competatief is met een commercieel tool.

- Hoofdstuk 7: *Model Checking the Time to Reach Agreement*. In dit hoofdstuk wordt een typisch verificatieprobleem gepresenteerd dat zeer moeilijk is voor model checkers en dat slechts drie jaar geleden nog onmogelijk te analyseren was.

Conclusies

Het is al jaren duidelijk dat model checking kan bijdragen aan het ontwerp en de analyse van realistische systemen. Veel onderzoek was en is gericht op het fundamentele schaalbaarheidsprobleem. Dit heeft geresulteerd in zeer krachtige tools die het experts op het gebied van formele verificatie mogelijk maken om vele interessante en realistische case studies routinematig op te lossen. De ASML-case study uit hoofdstuk 5 en de Agreement-case study uit hoofdstuk 7 illustreren dit.

Het feit dat vele technieken die gericht zijn op schaalbaarheid vaak tegelijk worden toegepast, roept vragen op over correctheid van zulke combinaties. Buiten het feit dat de eigenlijke implementatie van het model checking algorithme correct moet zijn, moet het ook duidelijk zijn of de additionele technieken paarsgewijs compatibel zijn. Ook zorgen evoluerende modelleertalen voor problemen. De symmetriereductietechniek uit de hoofdstukken 3 en 4, bijvoorbeeld, is gebaseerd op een versie van UPPAAL die de C-achtige taal uit de huidige versie nog niet ondersteunde. Hoe moet hiermee om worden gegaan? Moet al het theoretische werk weer opnieuw worden gedaan om correctheid te garanderen?

Ondanks dat tenminste twee van de drie case studies uit dit proefschrift snel en routinematig zijn opgelost, laten de case studies ook duidelijk zien dat model checking nog geen “push-button technology” is. Er zijn drie problemen aan te wijzen (buiten het fundamentele schaalbaarheidsprobleem):

- Het logisch en rechttoe-rechtaan modelleren van het probleem kan makkelijk resulteren in een model dat te gedetailleerd is om te kunnen analyseren.

- Het is vaak erg moeilijk en onhandig om heuristieken aan het model toe te voegen.

- Soms is het door de beperkte kracht van de modelleertaal erg moeilijk om een bepaald aspect te modelleren.

De eerste van de bovengenoemde problemen is sterk gerelateerd aan het schaalbaarheidsprobleem en zal ook altijd een probleem blijven. Een effectieve manier om met dit probleem om te gaan is het construeren van abstracties die precies zo gedetailleerd zijn als nodig is. Dit is echter een niet triviale bezigheid zoals duidelijk wordt geïllustreerd door de Agreement-case study. De ASML-case study geeft een mooi voorbeeld van het gebruik van abstractie. Het verificatieprobleem (waarvoor de hele toestandsruimte doorgerekend moet worden) wordt opgelost met behulp van een abstract model en het optimalisatieprobleem wordt door middel van heuristieken en een veel concreter model opgelost. Bovendien wordt handmatig bewezen dat de abstractie correct is. Idealiter behoeft het proces van het construeren en correct bewijzen van abstracties geen of slechts zeer weinig menselijke interactie.

Het tweede probleem is typisch van toepassing op scheduling problemen. In het algemeen is de toestandsruimte van deze problemen veel te groot om helemaal door te rekenen. Daarom worden vaak heuristieken gebruikt om snel goede schedules te vinden. Het modelleren van heuristieken is vaak niet eenvoudig. Dit is duidelijk een toegankelijkheidsprobleem. Bovendien is het in het algemeen niet gemakkelijk om de heuristieken op een elegante wijze van de kern van het model te scheiden. Dit heeft als gevolg dat er snel vele versies van modellen ontstaan, wat de onderhoudbaarheid van het model niet bevordert.

Het derde probleem wordt geïllustreerd door de werktijden van het personeel in de AXXOM-case study. Het modelleren van deze werktijden met geklokte automaten is niet natuurlijk en heeft veel voeten in de aarde. Dit is weer een typisch toegankelijkheidsprobleem: het modelleren is moeilijk en er moet bijzonder goed worden opgepast dat het model analyseerbaar blijft. De conclusie is hier dan ook dat pure geklokte automaten – net als ieder ander low level formalisme – niet erg geschikt zijn om dit soort high level constraints te modelleren.

Het tweede en derde probleem zou kunnen worden ondervangen door high level domein-specifieke talen die een front end voor geklokte automaten modellen vormen. In huidig werk wordt deze aanpak onderzocht voor de exploratie van de ontwerpruimte van embedded systeemarchitecturen [61]. Een alternatief wordt gevormd door het aanbieden van een bibliotheek met *problem templates* voor specifieke applicatiedomeinen. De gebruiker selecteert een template die erg dicht bij zijn eigen probleem ligt (inclusief nuttige heuristieken). In het ideale geval hoeven er slechts marginale veranderingen te worden uitgevoerd om het model geschikt te maken voor het nieuwe probleem. Is dit echter niet het geval dan is gedetailleerde kennis van geklokte automaten toch weer noodzakelijk.

Het fundamentele schaalbaarheidsprobleem blijft altijd aanwezig en er zal altijd intensief onderzoek nodig zijn om de grenzen op te schuiven. Het AMETIST-project heeft grote vooruitgang geboekt. De performance van UPPAAL is gedurende dit driejarige project met verscheidene ordes vergroot. Er bestaan echter nog vele technieken die aan UPPAAL zouden kunnen worden toegevoegd om het tool nog sneller te maken. Deze zijn bijvoorbeeld de reductie van "inactieve integer variabelen" [106], klokoptimalisatie [40], "slicing" gebaseerd op de verifi-

catie property [29], de "sweep-line" methode[12] [36], en alternatieve symbolische technieken zoals bijvoorbeeld gepresenteerd worden in [103]. De problemen van toegankelijkheid en gebruiksgemak lijken steeds meer zichtbaar te worden met het groeien van de performance van model checking tools. Een manier om deze op te lossen is om high level talen voor specifieke applicatiedomeinen te construeren die zich laten vertalen naar efficiënte low level modellen die geanalyseerd kunnen worden met de bestaande tools. Een andere manier is het vullen van een "case study bibliotheek" waaruit reeds opgeloste problemen kunnen worden geleend. Deze kunnen vervolgens worden aangepast om het bestaande probleem te analyseren.

[12] Recent is begonnen om deze techniek aan UPPAAL toe te voegen.

Curriculum Vitae

Martijn Hendriks was born in Canberra, Australia, on August 20, 1976. He lived there for less than one year and then moved to Nijmegen, The Netherlands. After graduation from the Elshof College he started his Chemistry studies in Nijmegen in 1994. After 4 years he switched to Computer Science, also in Nijmegen. In February 2002 he graduated cum laude and became a PhD student in the group Informatics for Technical Applications, headed by prof. dr. Frits W. Vaandrager. Currently, he works as a researcher in the same group.

Titles in the IPA Dissertation Series

J.O. Blanco. *The State Operator in Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1996-01

A.M. Geerling. *Transformational Development of Data-Parallel Algorithms*. Faculty of Mathematics and Computer Science, KUN. 1996-02

P.M. Achten. *Interactive Functional Programs: Models, Methods, and Implementation*. Faculty of Mathematics and Computer Science, KUN. 1996-03

M.G.A. Verhoeven. *Parallel Local Search*. Faculty of Mathematics and Computing Science, TUE. 1996-04

M.H.G.K. Kesseler. *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory*. Faculty of Mathematics and Computer Science, KUN. 1996-05

D. Alstein. *Distributed Algorithms for Hard Real-Time Systems*. Faculty of Mathematics and Computing Science, TUE. 1996-06

J.H. Hoepman. *Communication, Synchronization, and Fault-Tolerance*. Faculty of Mathematics and Computer Science, UvA. 1996-07

H. Doornbos. *Reductivity Arguments and Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1996-08

D. Turi. *Functorial Operational Semantics and its Denotational Dual*. Faculty of Mathematics and Computer Science, VUA. 1996-09

A.M.G. Peeters. *Single-Rail Handshake Circuits*. Faculty of Mathematics and Computing Science, TUE. 1996-10

N.W.A. Arends. *A Systems Engineering Specification Formalism*. Faculty of Mechanical Engineering, TUE. 1996-11

P. Severi de Santiago. *Normalisation in Lambda Calculus and its Relation to Type Inference*. Faculty of Mathematics and Computing Science, TUE. 1996-12

D.R. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. Faculty of Mathematics and Computing Science, TUE. 1996-13

M.M. Bonsangue. *Topological Dualities in Semantics*. Faculty of Mathematics and Computer Science, VUA. 1996-14

B.L.E. de Fluiter. *Algorithms for Graphs of Small Treewidth*. Faculty of Mathematics and Computer Science, UU. 1997-01

W.T.M. Kars. *Process-algebraic Transformations in Context*. Faculty of Computer Science, UT. 1997-02

P.F. Hoogendijk. *A Generic Theory of Data Types*. Faculty of Mathematics and Computing Science, TUE. 1997-03

T.D.L. Laan. *The Evolution of Type Theory in Logic and Mathematics*. Faculty of Mathematics and Computing Science, TUE. 1997-04

C.J. Bloo. *Preservation of Termination for Explicit Substitution*. Faculty of Mathematics and Computing Science, TUE. 1997-05

J.J. Vereijken. *Discrete-Time Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1997-06

F.A.M. van den Beuken. *A Functional Approach to Syntax and Typing*. Faculty of Mathematics and Informatics, KUN. 1997-07

A.W. Heerink. *Ins and Outs in Refusal Testing*. Faculty of Computer Science, UT. 1998-01

G. Naumoski and W. Alberts. *A Discrete-Event Simulator for Systems Engineering*. Faculty of Mechanical Engineering, TUE. 1998-02

J. Verriet. *Scheduling with Communication for Multiprocessor Computation*. Faculty of Mathematics and Computer Science, UU. 1998-03

J.S.H. van Gageldonk. *An Asynchronous Low-Power 80C51 Microcontroller*. Faculty of Mathematics and Computing Science, TUE. 1998-04

A.A. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. Faculty of Mathematics and Computing Science, TUE. 1998-05

E. Voermans. *Inductive Datatypes with Laws and Subtyping – A Relational Model*. Faculty of Mathematics and Computing Science, TUE. 1999-01

H. ter Doest. *Towards Probabilistic Unification-based Parsing*. Faculty of Computer Science, UT. 1999-02

J.P.L. Segers. *Algorithms for the Simulation of Surface Processes*. Faculty of Mathematics and Computing Science, TUE. 1999-03

C.H.M. van Kemenade. *Recombinative Evolutionary Search*. Faculty of Mathematics and Natural Sciences, UL. 1999-04

E.I. Barakova. *Learning Reliability: a Study on Indecisiveness in Sample Selection*. Faculty of Mathematics and Natural Sciences, RUG. 1999-05

M.P. Bodlaender. *Scheduler Optimization in Real-Time Distributed Databases*. Faculty of Mathematics and Computing Science, TUE. 1999-06

M.A. Reniers. *Message Sequence Chart: Syntax and Semantics*. Faculty of Mathematics and Computing Science, TUE. 1999-07

J.P. Warners. *Nonlinear approaches to satisfiability problems*. Faculty of Mathematics and Computing Science, TUE. 1999-08

J.M.T. Romijn. *Analysing Industrial Protocols with Formal Methods*. Faculty of Computer Science, UT. 1999-09

P.R. D'Argenio. *Algebras and Automata for Timed and Stochastic Systems*. Faculty of Computer Science, UT. 1999-10

G. Fábián. *A Language and Simulator for Hybrid Systems*. Faculty of Mechanical Engineering, TUE. 1999-11

J. Zwanenburg. *Object-Oriented Concepts and Proof Rules*. Faculty of Mathematics and Computing Science, TUE. 1999-12

R.S. Venema. *Aspects of an Integrated Neural Prediction System*. Faculty of Mathematics and Natural Sciences, RUG. 1999-13

J. Saraiva. *A Purely Functional Implementation of Attribute Grammars*. Faculty of Mathematics and Computer Science, UU. 1999-14

R. Schiefer. *Viper, A Visualisation Tool for Parallel Program Construction*. Faculty of Mathematics and Computing Science, TUE. 1999-15

K.M.M. de Leeuw. *Cryptology and Statecraft in the Dutch Republic*. Faculty of Mathematics and Computer Science, UvA. 2000-01

T.E.J. Vos. *UNITY in Diversity. A stratified approach to the verification of distributed algorithms*. Faculty of Mathematics and Computer Science, UU. 2000-02

W. Mallon. *Theories and Tools for the Design of Delay-Insensitive Communicating Processes*. Faculty of Mathematics and Natural Sciences, RUG. 2000-03

W.O.D. Griffioen. *Studies in Computer Aided Verification of Protocols*. Faculty of Science, KUN. 2000-04

P.H.F.M. Verhoeven. *The Design of the MathSpad Editor*. Faculty of Mathematics and Computing Science, TUE. 2000-05

J. Fey. *Design of a Fruit Juice Blending and Packaging Plant*. Faculty of Mechanical Engineering, TUE. 2000-06

M. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. Faculty of Mathematics and Computing Science, TUE. 2000-07

P.A. Olivier. *A Framework for Debugging Heterogeneous Applications*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08

E. Saaman. *Another Formal Specification Language*. Faculty of Mathematics and Natural Sciences, RUG. 2000-10

M. Jelasity. *The Shape of Evolutionary Search Discovering and Representing Search Space Structure*. Faculty of Mathematics and Natural Sciences, UL. 2001-01

R. Ahn. *Agents, Objects and Events a computational approach to knowledge, observation and communication*. Faculty of Mathematics and Computing Science, TU/e. 2001-02

M. Huisman. *Reasoning about Java programs in higher order logic using PVS and Isabelle*. Faculty of Science, KUN. 2001-03

I.M.M.J. Reymen. *Improving Design Processes through Structured Reflection*. Faculty of Mathematics and Computing Science, TU/e. 2001-04

S.C.C. Blom. *Term Graph Rewriting: syntax and semantics*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05

R. van Liere. *Studies in Interactive Visualization*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06

A.G. Engels. *Languages for Analysis and Testing of Event Sequences*. Faculty of Mathematics and Computing Science, TU/e. 2001-07

J. Hage. *Structural Aspects of Switching Classes*. Faculty of Mathematics and Natural Sciences, UL. 2001-08

M.H. Lamers. *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes*. Faculty of Mathematics and Natural Sciences, UL. 2001-09

T.C. Ruys. *Towards Effective Model Checking*. Faculty of Computer Science, UT. 2001-10

D. Chkliaev. *Mechanical verification of concurrency control and recovery protocols*. Faculty of Mathematics and Computing Science, TU/e. 2001-11

M.D. Oostdijk. *Generation and presentation of formal mathematical documents*. Faculty of Mathematics and Computing Science, TU/e. 2001-12

A.T. Hofkamp. *Reactive machine control: A simulation approach using χ*. Faculty of Mechanical Engineering, TU/e. 2001-13

D. Bošnački. *Enhancing state space reduction techniques for model checking*. Faculty of Mathematics and Computing Science, TU/e. 2001-14

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06